PROFESSOR: All right, well, we've seen how the query language works. Now, let's talk about how it's implemented. You already pretty much can guess what's going on there. At the bottom of it, there's a pattern matcher. And we looked at a pattern matcher when we did the rule-based control language.

Just to remind you, here are some sample patterns. This is a pattern that will match any list of three things of which the first is a and the second is c and the middle one can be anything. So in this little pattern-matching syntax, there's only one distinction you make. There's either literal things or variables, and variables begin with question mark.

So this matches any list of three things of which the first is a and the second is c. This one matches any list of three things of which the first is the symbol job. The second can be anything. And the third is a list of two things of which the first is the symbol computer and the second can be anything. And this one, this next one matches any list of three things, and the only difference is, here, the third list, the first is the symbol computer, and then there's some rest of the list. So this means two elements and this means arbitrary number. And our language implementation isn't even going to have to worry about implementing this dot because that's automatically done by Lisp's reader.

Remember matchers also have some consistency in them. This match is a list of three things of which the first is a. And the second and third can be anything, but they have to be the same thing. They're both called x. And this matches a list of four things of which the first is the fourth and the second is the same as the third. And this last one matches any list that begins with a. The first thing is a, and the rest can be anything. So that's just a review of pattern matcher syntax that you've already seen.

And remember, that's implemented by some procedure called match. And match takes a pattern and some data and a dictionary. And match asks the question is there any way to match this pattern against this data object subject to the bindings that are already in this dictionary?

So, for instance, if we're going to match the pattern x, y, y, x against the data a, b, b, a subject to a dictionary, that says x equals a. Then the matcher would say, yes, that's consistent. These match, and it's consistent with what's in the dictionary to say that x equals a. And the result of the match is the extended dictionary that says x equals a and y equals b. So a matcher takes in pattern data dictionary, puts out an extended dictionary if it matches, or if it doesn't match, says that it fails. So, for example, if I use the same pattern here, if I say this x, y, y, x match a, b, b, a with the dictionary y equals a, then the matcher would put out fail.

Well, you've already seen the code for a pattern matcher so I'm not going to go over it, but it's the same thing

we've been doing before. You saw that in the system on rule-based control. It's essentially the same matcher. In fact, I think the syntax is a little bit simpler because we're not worrying about arbitrary constants and expressions and things. There's just variables and constants.

OK, well, given that, what's a primitive query? Primitive query is going to be a rather complicated thing. It's going to be-- let's think about the query job of x is d dot y. That's a query we might type in. That's going to be implemented in the system. We'll think of it as this little box. Here's the primitive query.

What this little box is going to do is take in two streams and put out a stream. So the shape of a primitive query is that it's a thing where two streams come in and one stream goes out. What these streams are going to be is down here is the database. So we imagine all the things in the database sort of sitting there in a stream and this thing sucks on them. So what are some things that might be in the database? Oh, job of Alyssa is something and some other job is something. So imagine all of the facts in the database sitting there in the stream. That's what comes in here.

What comes in here is a stream of dictionaries. So one particular dictionary might say y equals programmer. Now, what the query does when it gets in a dictionary from this stream, it finds all possible ways of matching the query against whatever is coming in from the database. It looks at the query as a pattern, matches it against any fact from the database or all possible ways of finding and matching the database with respect to this dictionary that's coming in. So for each fact in the database, it calls the matcher using the pattern, fact, and dictionary. And every time it gets a good match, it puts out the extended dictionary.

So, for example, if this one comes in and it finds a match, out will come a dictionary that in this case will have y equals programmer and x equals something. y is programmer, x is something, and d is whatever it found. And that's all. And, of course, it's going to try this for every fact in the dictionary. So it might find lots of them. It might find another one that says y equals programmer and x equals, and d equals.

So for one frame coming in, it might put out-- for one dictionary coming in, it might put out a lot of dictionaries, or it might put out none. It might have something that wouldn't match like x equals FOO. This one might not match anything in which case nothing will go into this stream corresponding to this frame. Or what you might do is put in an empty frame, and an empty frame says try matching all ways-- find all possible ways of matching the query against something in the database subject to no previous restrictions. And if you think about what that means, that's just the computation that's done when you type in a query right off. It tries to find all matches.

So a primitive query sets up this mechanism. And what the language does, when you type in the query at the top level, it takes this mechanism, feeds in one single empty dictionary, and then for each thing that comes out takes

the original query and instantiates the result with all the different dictionaries, producing a new stream of instantiated patterns here. And that's what gets printed on the terminal. That's the basic mechanism going on there.

Well, why is that so complicated? You probably can think of a lot simpler ways to arrange this match for a primitive query rather than having all of these streams floating around. And the answer is-- you probably guess already. The answer is this thing extends elegantly to implement the means of combination. So, for instance, suppose I don't only want to do this. I don't want to say who to be everybody's job description.

Suppose I want to say AND the job of x is d dot y and the supervisor of x is z. Now, supervisor of x is z is going to be another primitive query that has the same shape to take in a stream of data objects, a stream of initial dictionaries, which are the restrictions to try and use when you match, and it's going to put out a stream of dictionaries. So that's what this primitive query looks like. And how do I implement the AND? Well, it's simple. I just hook them together. I take the output of this one, and I put that to the input of that one. And I take the dictionary here and I fan it out.

And then you see how that's going to work, because what's going to happen is a frame will now come in here, which has a binding for x, y, and d. And then when this one gets it, it'll say, oh, gee, subject to these restrictions, which now already have values in the dictionary for y and x and d, it looks in the database and says, gee, can I find any supervisor facts? And if it finds any, out will come dictionaries which have bindings for y and x and d and z now.

And then notice that because the frames coming in here have these restrictions, that's the thing that assures that when you do the AND, this x will mean the same thing as that x. Because by the time something comes floating in here, x has a value that you have to match against consistently. And then you remember from the code from the matcher, there was something in the way the matcher did dictionaries that arrange consistent matches. So there's AND.

The important point to notice is the general shape. Look at what happened: the AND of two queries, say, P and Q. Here's P and Q. The AND of two queries, well, it looks like this. Each query takes in a stream from the database, a stream of inputs, and puts out a stream of outputs. And the important point to notice is that if I draw a box around this thing and say this is AND of P and Q, then that box has exactly the same overall shape. It's something that takes in a stream from the database. Here it's going to get fanned out inside, but from the outside you don't see that. It takes an input stream and puts out an output stream.

So this is AND. And then similarly, OR would look like this. OR would-- although I didn't show you examples of OR. OR would say can I find all ways of matching P or Q. So I have P and Q. Each will have their shape. And the way

OR is implemented is I'll take my database stream. I'll fan it out. I'll put one into P and one into Q. I'll take my initial query stream coming in and fan it out. So I'll look at all the answers I might get from P and all the answers I might get from Q, and I'll put them through some sort of thing that appends them or merges the result into one stream, and that's what will come out. And this whole thing from the outside is OR. And again, you see it has the same overall shape when looked at from the outside.

What's NOT? NOT works kind of the same way. If I have some query P, I take the primitive query for P. Here, I'm going to implement NOT P. And NOT's just going to act as a filter. I'll take in the database and my original stream of dictionaries coming in, and what NOT P will do is it will filter these guys. And the way it will filter it, it will say when I get in a dictionary here, I'll find all the matches, and if I find any, I'll throw it away. And if I don't find any matches to something coming in here, I'll just pass that through, so NOT is a pure filter.

So AND is-- think of these sort of electoral resistors or something. AND is series combination and OR is parallel combination. And then NOT is not going to extend any dictionaries at all. It's just going to filter it. It's going to throw away the ones for which it finds a way to match. And list value is sort of the same way. The filter's a little more complicated. It applies to predicate.

The major point to notice here, and it's a major point we've looked at before, is this idea of closure. The things that we build as a means of combination have the same overall structure as the primitive things that we're combining. So the AND of two things when looked at from the outside has the same shape. And what that means is that this box here could be an AND or an OR or a NOT or something because it has the same shape to interface to the larger things.

It's the same thing that allowed us to get complexity in the Escher picture language or allows you to immediately build up these complicated structures just out of pairs. It's closure. And that's the thing that allowed me to do what by now you took for granted when I said, gee, there's a query which is AND of job and salary, and I said, oh, there's another one, which is AND of job, a NOT of something. The fact that I can do that is a direct consequence of this closure principle. OK, let's break and then we'll go on.

AUDIENCE: Where does the dictionary come from?

PROFESSOR: The dictionary comes initially from what you type in. So when you start this up, the first thing it does is set up this whole structure. It puts in one empty dictionary. And if all you have is one primitive query, then what will come out is a bunch of dictionaries with things filled in. The general situation that I have here is when this is in the middle of some nest of combined things.

Let's look at the picture over here. This supervisor query gets in some dictionary. Where did this one come from?

This dictionary came from the fact that I'm looking at the output of this primitive query. So maybe to be very specific, if I literally typed in just this query at the top level, this AND, what would actually happen is it would build this structure and start up this whole thing with one empty dictionary. And now this one would process, and a whole bunch of dictionaries would come out with x, y's and d's in them. Run it through this one.

So now that's the input to this one. This one would now put out some other stuff. And if this itself were buried in some larger thing, like an OR of something, then that would go feed into the next one. So you initially get only one empty dictionary when you start it, but as you're in the middle of processing these compounds things, that's where these cascades of dictionaries start getting generated.

AUDIENCE: Dictionaries only come about as a result of using the queries? Or do they become-- do they stay someplace in space like the database does? Are these temporary items?

PROFESSOR: They're created temporarily in the matcher. Really, they're someplace in storage. Initially, someone creates a thing called the empty dictionary that gets initially fed to this match procedure, and then the match procedure builds some dictionaries, and they get passed on and on.

AUDIENCE: OK, so they'll go way after the match?

PROFESSOR: They'll go away when no one needs them again, yeah.

AUDIENCE: It appears that the AND performs some redundant searches of the database. If the first clause matched, let's say, the third element and not on the first two elements, the second clause is going to look at those first two elements again, discarding them because they don't match. The match is already in the dictionary. Would it makes sense to carry the data element from the database along with the dictionary?

PROFESSOR: Well, in general, there are other ways to arrange this search, and there's some analysis that you can do. I think there's a problem in the book, which talks about a different way that you can cascade AND to eliminate various kinds of redundancies. This one is meant to be-- was mainly meant to be very simple so you can see how they fit together. But you're quite right. There are redundancies here that you can get rid of. That's another reason why this language is somewhat slow. There are a lot smarter things you can do. We're just trying to show you a very simple, in principle, implementation.

AUDIENCE: Did you model this language on Prolog, or did it just come out looking like Prolog?

PROFESSOR: Well, Jerry insulted a whole bunch of people yesterday, so I might as well say that the MIT attitude towards Prolog is something that people did in about 1971 and decided that it wasn't really the right thing and stopped. So we modeled this on the sort of natural way that this thing was done in about 1971, except at that

point, we didn't do it with streams. After we were using it for about six months, we discovered that it had all these problems, some of which I'll talk about later. And we said, gee, Prolog must have fixed those, and then we found out that it didn't. So this does about the same thing as Prolog.

AUDIENCE: Does Prolog use streams?

PROFESSOR: No. In how it behaves, it behaves a lot like Prolog. Prolog uses a backtracking strategy. But the other thing that's really good about Prolog that makes it a usable thing is that there's a really very, very well-engineered compiler technology that makes it run fast. So although you saw the merge spitting out these answers very, very slowly, a real Prolog will run very, very fast. Because even though it's sort of doing this, the real work that went into Prolog is a very, very excellent compiler effort. Let's take a break.

We've looked at the primitive queries and the ways that streams are used to implement the means of combination: AND and OR and NOT. Now, let go on to the means of abstraction. Remember, the means of abstraction in this language are rules. So z is a boss in division d if there's some x who has a job in division d and z is the supervisor of x. That's what it means for someone to be a boss.

And in effect, if you think about what we're doing with relation to this, there's the query we wrote-- the job of x is in d and the supervisor of x is z-- what we in effect want to do is take this whole mess and draw a box around it and say this whole thing inside the box is boss of z in division d. That's in effect what we want to do.

So, for instance, if we've done that, and we want to check whether or not it's true that Ben Bitdiddle is a boss in the computer division, so if I want to say boss of Ben Bitdiddle in the computer division, imagine typing that in as query to the system, in effect what we want to do is set up a dictionary here, which has z to Ben Bitdiddle and d to computer. Where did that dictionary come from? Let's look at the slide for one second.

That dictionary came from matching the query that said boss of Ben Bitdiddle and computer onto the conclusion of the rule: boss of z and d. So we match the query to the conclusion of the rule. That gives us a dictionary, and that's the thing that we would now like to put into this whole big thing and process and see if anything comes out the other side. If anything comes out, it'll be true. That's the basic idea.

So in general, the way we implement a rule is we match the conclusion of the rule against something we might want to check it's true. That match gives us a dictionary, and with respect to that dictionary, we process the body of the rule.

Well, that's really all there is, except for two technical points. The first technical point is that I might have said something else. I might have said who's the boss in the computer division? So I might say boss of who in computer division. And if I did that, what I would really like to do in effect is start up this dictionary with a match

that sort of says, well, d is computer and z is whatever who is. And our matcher won't quite do that. That's not quite matching a pattern against data. It's matching two patterns and saying are they consistent or not or what ways make them consistent.

In other words, what we need is not quite a pattern matcher, but something a little bit more general called a unifier. And a unifier is a slight generalization of a pattern matcher. What a unifier does is take two patterns and say what's the most general thing you can substitute for the variables in those two patterns to make them satisfy the pattern simultaneously? Let me give you an example.

If I have the pattern two-element list, which is x and x, so I have a two-element list where both elements are the same and otherwise I don't care what they are, and I unify that against the pattern that says there's a two-element list, and the first one is a and something in c and the second one is a and b and z, then what the unifier should tell me is, oh yeah, in that dictionary, x has to be a, b, c, and y has to be d and z has to be c. Those are the restrictions I'd have to put on the values of x, y, and z to make these two unify, or in other words, to make this match x and make this match x. The unifier should be able to deduce that.

But the unifier may-- there are more complicated things. I might have said something a little bit more complicated. I might have said there's a list with two elements, and they're both the same, and they should unify against something of this form. And the unifier should be able to deduce from that. Like that y would have to be b. y would have to be b. Because these two are the same, so y's got to be b. And v here would have to be a. And z and w can be anything, but they have to be the same thing. And x would have to be b, followed by a, followed by whatever w is or whatever z is, which is the same. So you see, the unifier somehow has to deduce things to unify these patterns. So you might think there's some kind of magic deduction going on, but there's not.

A unifier is basically a very simple modification of a pattern matcher. And if you look in the book, you'll see something like three or four lines of code added to the pattern matcher you just saw to handle the symmetric case. Remember, the pattern matcher has a place where it says is this variable matching a constant. And if so, it checks in the dictionary. There's only one other clause in the unifier, which says is this variable matching a variable, in which case you go look in the dictionary and see if that's consistent with what's in the dictionary.

So all the, quote, deduction that's in this language, if you sort of look at it, sort of sits in the rule applications, which, if you look at that, sits in the unifier, which, if you look at that under a microscope, sits essentially in the pattern matcher. There's no magic at all going on in there. And the, quote, deduction that you see is just the fact that there's this recursion, which is unwinding the matches bit by bit. So it looks like this thing is being very clever, but in fact, it's not being very clever at all.

There are cases where a unifier might have to be clever. Let me show you one more. Suppose I want to unify a list of two elements, x and x, with a thing that says it's y followed by a dot y. Now, if you think of what that would have to mean, it would have to mean that x had better be the same as y, but also x had better be the same as a list whose first element is a and whose rest is y. And if you think about what that would have to mean, it would have to mean that y is the infinite list of a's. In some sense, in order to do that unification, I have to solve the fixed-point equation cons of a to y is equal to y.

And in general, I wrote a very simple one. Really doing unification might have to solve an arbitrary fixed-point equation: f of y equals y. And basically, you can't do that and make the thing finite all the time. So how does the logic language handle that? The answer is it doesn't. It just punts. And there's a little check in the unifier, which says, oh, is this one of the hard cases which when I go to match things would involve solving a fixed-point equation?

And in this case, I will throw up my hands. And if that check were not in there, what would happen? In most cases is that the unifier would just go into an infinite loop. And other logic programming languages work like that. So there's really no magic. The easy case is done in a matcher. The hard case is not done at all. And that's about the state of this technology.

Let me just say again formally how rules work now that I talked about unifiers. So the official definition is that to apply a rule, we-- well, let's start using some words we've used before. Let's talk about sticking dictionaries into these big boxes of query things as evaluating these large queries relative to an environment or a frame. So when you think of that dictionary, what's the dictionary after all? It's a bunch of meanings for symbols. That's what we've been calling frames or environments.

What does it mean to do some processing relevant to an environment? That's what we've been calling evaluation. So we can say the way that you apply a rule is to evaluate the rule body relative to an environment that's formed by unifying the rule conclusion with the given query. And the thing I want you to notice is the complete formal similarity to the net of circular evaluator or the substitution model.

To apply a procedure, we evaluate the procedure body relative to an environment that's formed by blinding the procedure parameters to the arguments. There's a complete formal similarity here between the rules, rule application, and procedure application even though these things are very, very different. And again, you have the EVAL APPLY loop. EVAL and APPLY.

So in general, I might be processing some combined expression that will turn into a rule application, which will generate some dictionaries or frames or environments-- whatever you want to call them-- from match, which will then be the input to some big compound thing like this. This has pieces of it and may have other rule applications.

And you have essentially the same cycle even though there's nothing here at all that looks like procedures. It really has to do with the fact you've built a language whose means of combination and abstraction unwind in certain ways.

And then in general, what happens at the very top level, you might have rules in your database also, so things in this database might be rules. There are ways to check that things are true. So it might come in here and have to do a rule check. And then there's some control structure which says, well, you look at some rules, and you look at some data elements, and you look at some rules and data elements, and these fan out and out and out.

So it becomes essentially impossible to say what order it's looking at these things in, whether it's breadth first or depth first or anything. And it's even more impossible because the actual order is somehow buried in the delays of the streams. So what's very hard to tell from this is the order in which it's scanned. But what's true, because you're looking at the stream view, is that all of them eventually get looked at.

Let me just mention one tiny technical problem. Suppose I tried saying boss of y is computer, then a funny thing would happen. As I stuck a dictionary with y in here, I might get-- this y is not the same as that y, which was the other piece of somebody's job description. So if I really only did literally what I said, we'd get some variable conflict problems. So I lied to you a little bit.

Notice that problem is exactly a problem we've run into before. It is precisely the need for local variables in a language. When I have the sum of squares, that x had better not be that x. That's exactly the same as this y had better not be that y. And we know how to solve that. That was this whole environment model, and we built chains of frames and all sorts of things like that.

There's a much more brutal way to solve it. In the query language, we didn't even do that. We did something completely brutal. We said every time you apply a rule, rename consistently all the variables in the rule to some new unique names that won't conflict with anything. That's conceptually simpler, but really brutal and not particularly efficient.

But notice, we could have gotten rid of all of our environment structures if we defined for procedures in Lisp the same thing. If every time we applied a procedure and did the substitution model we renamed all the variables in the procedure, then we never would have had to worry about local variables because they would never arise. OK, well, that would be inefficient, and it's inefficient here in the query language, too, but we did it to keep it simple. Let's break for questions.

AUDIENCE: When you started this section, you emphasized how powerful our APPLY EVAL model was that we could use it for any language. And then you say we're going to have this language which is so different. It turns

out that this language, as you just pointed out, is very much the same. I'm wondering if you're arguing that all languages end up coming down to this you can apply a rule or apply a procedure or some kind of apply?

PROFESSOR: I would say that pretty much any language where you really are building up these means of combination and giving them simpler names and you're saying anything of the sort, like here's a general kind of expression, like how to square something, almost anything that you would call a procedure. If that's got to have parts, you have to unwind those parts. You have to have some kind of organization which says when I look at the abstract variables or tags or whatever you want to call them that might stand for particular things, you have to keep track of that, and that's going to be something like an environment. And then if you say this part can have parts which I have to unwind, you've got to have something like this cycle.

And lots and lots of languages have that character when they sort of get put together in this way. This language again really is different because there's nothing like procedures on the outside. When you go below the surface and you see the implementation, of course, it starts looking the same. But from the outside, it's a very different world view. You're not computing functions of inputs.

AUDIENCE: You mentioned earlier that when you build all of these rules in pattern matcher and with the delayed action of streams, you really have no way to know in what order things are evaluated.

PROFESSOR: Right.

AUDIENCE: And that would indicate then that you should only express declarative knowledge that's true for all-time, no-time sequence built into it. Otherwise, these things get all--

PROFESSOR: Yes. Yes. The question is this really is set up for doing declarative knowledge, and as I presented it-- and I'll show you some of the ugly warts under this after the break. As I presented it, it's just doing logic. And in principle, if it were logic, it wouldn't matter what order it's getting done. And it's quite true when you start doing things where you have side effects like adding things to the database and taking things out, and we'll see some others, you use that kind of control.

So, for example, contrasting with Prolog. Say Prolog has various features where you really exploit the order of evaluation. And people write Prolog programs that way. That turns out to be very complicated in Prolog, although if you're an expert Prolog programmer, you can do it. However, here I don't think you can do it at all. It's very complicated because you really are giving up control over any prearranged order of trying things.

AUDIENCE: Now, that would indicate then that you have a functional mapping. And when you started out this lecture, you said that we express the declarative knowledge which is a relation, and we don't talk about the inputs and the outputs.

PROFESSOR: Well, there's a pun on functional, right? There's function in the sense of no side effects and not depending on what order is going on. And then there's functional in the sense of mathematical function, which means input and output. And it's just that pun that you're making, I think.

AUDIENCE: I'm a little unclear on what you're doing with these two statements, the two boss statements. Is the first one building up the database and the second one a query or--

PROFESSOR: OK, I'm sorry. What I meant here, if I type something like this in as a query-- I should have given an example way at the very beginning. If I type in job, Ben Bitdiddle, computer wizard, what the processing will do is if it finds a match, it'll find a match to that exact thing, and it'll type out a job, Ben Bitdiddle, computer wizard. If it doesn't find a match, it won't find anything.

So what I should have said is the way you use the query language to check whether something is true, remember, that's one of the things you want to do in logic programming, is you type in your query and either that comes out or it doesn't. So what I was trying to illustrate here, I wanted to start with a very simple example before talking about unifiers. So what I should have said, if I just wanted to check whether this is true, I could type that in and see if anything came out

AUDIENCE: And then the second one--

PROFESSOR: The second one would be a real query.

AUDIENCE: A real query, yeah.

PROFESSOR: What would come out, see, it would go in here say with FOO, and in would go frame that says z is bound to who and d is bound to computer. And this will pass through, and then by the time it got out of here, who would pick up a binding.

AUDIENCE: On the unifying thing there, I still am not sure what happens with who and z. If the unifying-- the rule here says-- OK, so you say that you can't make question mark equal to question mark who.

PROFESSOR: Right. That's what the matcher can't do. But what this will mean to a unifier is that there's an environment with three variables. d here is computer. z is whatever who is. So if later on in the matcher routine it said, for example, who has to be 3, then when I looked up in the dictionary, it will say, oh, z is 3 because it's the same as who. And that's in some sense the only thing you need to do to extend the unifier to a matcher.

AUDIENCE: OK, because it looked like when you were telling how to unify it, it looked like you would put the things together in such a way that you'd actually solve and have a value for both of them. And what it looks like now is

that you're actually pass a dictionary with two variables and the variables are linked.

PROFESSOR: Right. It only looks like you're solving for both of them because you're sort of looking at the whole solution at once. If you sort of watch the thing getting built up recursively, it's merely this.

AUDIENCE: OK, so you do pass off that dictionary with two variables?

PROFESSOR: That's right.

AUDIENCE: And link?

PROFESSOR: Right. It just looks like an ordinary dictionary.

AUDIENCE: When you're talking about the unifier, is it that there are some cases or some points that you are not able to use by them?

PROFESSOR: Right.

AUDIENCE: Can you just by building the rules or writing the forms know in advance if you are going to be able to solve to get the unification or not? Can you add some properties either to the rules itself or to the formula that you're writing so that you avoid the problem of not finding unification?

PROFESSOR: I mean, you can agree, I think, to write in a fairly restricted way where you won't run into it. See, because what you're getting-- see, the place where you get into problems is when you-- well, again, you're trying to match things like that against things where these have structure, where a, y, b, y something. So this is the kind of place where you're going to get into trouble.

AUDIENCE: So you can do that syntactically?

PROFESSOR: So you can kind of watch your rules in the kinds of things that your writing.

AUDIENCE: So that's the problem that the builder of the database has to be concerned?

PROFESSOR: That's a problem. It's a problem either-- not quite the builder of the database, the person who is expressing the rules, or the builder of the database. What the unifier actually does is you can check at the next level down when you actually get to the unifier and you'll see in the code where it looks up in the dictionary. If it sort of says what does y have to be? Oh, does y have to be something that contains a y as its expression? At that point, the unifier and say, oh my God, I'm trying to solve a fixed-point equation. I'll give it up here.

AUDIENCE: You make the distinction between the rules in the database. Are the rules added to the database?

PROFESSOR: Yes. Yes, I should have said that. One way to think about rules is that they're just other things in the database. So if you want to check the things that have to be checked in the database, they're kind of virtual facts that are in the database.

AUDIENCE: But in that explanation, you made the differentiation between database and the rules itself.

PROFESSOR: Yeah, I probably should not have done that. The only reason to do that is in terms of the implementation. When you look at the implementation, there's a part which says check either primitive assertions in the database or check rules. And then the real reason why you can't tell what order things are going to come out in and is that the rules database and the data database sort of get merged in a kind of delayed evaluation way. And so that's what makes the order very complicated. OK, let's break.

We've just seen how the logic language works and how rules work. Now, let's turn to a more profound question.

What do these things mean? That brings us to the subtlest, most devious part of this whole query language business, and that is that it's not quite what it seems to be. AND and OR and NOT and the logical implication of rules are not really the AND and OR and NOT and logical implication of logic. Let me give you an example of that.

Certainly, if we have two things in logic, it ought to be the case that AND of P and Q is the same as AND of Q and P and that OR of P and Q is the same as OR of Q and P. But let's look here. Here's an example. Let's talk about somebody outranking somebody else in our little database organization.

We'll say s is outranked by b or if either the supervisor of this is b or there's some middle manager here, that supervisor of s is m, and m is outranked by b. So there's one way to define rule outranked by. Or we can write exactly the same thing, except at the bottom here, we reversed the order of these two clauses. And certainly if this were logic, those ought to mean the same thing.

However, in our particular implementation, if you say something like who's outranked by Ben Bitdiddle, what you'll find is that this rule will work perfectly well and generate answers, whereas this rule will go into an infinite loop. And the reason for that is that this will come in and say, oh, who's outranked by Ben Bitdiddle? Find an s which is outranked by b, where b is Ben Bitdiddle, which is going to happen in it a subproblem. Oh gee, find an m such as m is outranked by Ben Bitdiddle with no restrictions on m.

So this will say in order to solve this problem, I solve exactly the same problem. And then after I've solved that, I'll check for a supervisory relationship. Whereas this one won't get into that, because before it tries to find this outranked by, it'll already have had a restriction on m here. So these two things which ought to mean the same, in fact, one goes into an infinite loop. One does not.

That's a very extreme case of a general thing that you'll find in logic programming that if you start changing the order of the things in the ANDs or ORs, you'll find tremendous differences in efficiency. And we just saw an infinitely big difference in efficiency and an infinite loop.

And there are similar things having to do with the order in which you enter rules. The order in which it happens to look at rules in the database may vastly change the efficiency with which it gets out answers or, in fact, send it into an infinite loop for some orderings. And this whole thing has to do with the fact that you're checking these rules in some order. And some rules may lead to really long paths of implication. Others might not. And you don't know a priori which ones are good and which ones are bad.

And there's a whole bunch of research having to do with that, mostly having to do with thinking about making parallel implementations of logic programming languages. And in some sense, what you'd like to do is check all rules in parallel and whichever ones get answers, you bubble them up. And if some go down infinite deductive changed, well, you just-- you know, memory is cheap and processors are cheap, and you just let them buzz for as for as long as you want.

There's a deeper problem, though, in comparing this logic language to real logic. The example I just showed you, it went into an infinite loop maybe, but at least it didn't give the wrong answer. There's an actual deeper problem when we start comparing, seriously comparing this logic language with real classical logic. So let's sort of review real classical logic.

All humans are mortal. That's pretty classical logic. Then maybe we'll continue in the very best classical tradition. We'll say all-- let's make it really classical. All Greeks are human, which has the syllogism that Socrates is a Greek. And then what do you write here? I think three dots, classical logic. Therefore, then the syllogism, Socrates is mortal. So there's some real honest classical logic.

Let's compare that with our classical logic database. So here's a classical logic database. Socrates is a Greek. Plato is a Greek. Zeus is a Greek, and Zeus is a god. And all humans are mortal. To show that something is mortal, it's enough to show that it's human. All humans are fallible. And all Greeks are humans is not quite right. This says that all Greeks who are not gods are human. So to show something's human, it's enough to show it's a Greek and not a god. And the address of any Greek god is Mount Olympus. So there's a little classical logic database. And indeed, that would work fairly well. If we type that in and say is Socrates mortal or Socrates fallible or mortal? It'll say yes. Is Plato mortal and fallible. It'll say yes. If we say is Zeus mortal? It won't find anything. And it'll work perfectly well.

However, suppose we want to extend this. Let's define what it means for someone to be a perfect being. Let's say

rule: a perfect being. And I think this is right. If you're up on your medieval scholastic philosophy, I believe that perfect beings are ones who were neither mortal nor fallible. AND NOT mortal x, NOT fallible x.

So we'll define this system to teach it what a perfect being is. And now what we're going to do is he ask for the address of all the perfect beings. AND the address of x is y and x is perfect. And so what we're generating here is the world's most exclusive mailing list. For the address of all the perfect things, we might have typed this in. Or we might type in this. We'll say AND perfect of x and the address of x is y.

Well, suppose we type all that in and we try this query. This query is going to give us an answer. This query will say, yeah, Mount Olympus. This query, in fact, is going to give us nothing. It will say no addresses of perfect beings.

Now, why is that? Why is there a difference? This is not an infinite loop question. This is a different answer question. The reason is that if you remember the implementation of NOT, NOT acted as a filter. NOT said I'm going to take some possible dictionaries, some possible frames, some possible answers, and filter out the ones that happened to satisfy some condition, and that's how I implement NOT. If you think about what's going on here, I'll build this query box where the output of an address piece gets fed into a perfect piece. What will happen is the address piece will set up some things of everyone whose address I know. Those will get filtered by the NOTs inside perfect here. So it will throw out the ones which happened to be either mortal or fallible.

In the other order what happens is I set this up, started up with an empty frame. The perfect in here doesn't find anything for the NOTs to filter, so nothing comes out here at all. And there's sort of nothing there that gets fed into the address thing. So here, I don't get an answer. And again, the reason for that is NOT isn't generating anything. NOT's only throwing out things. And if I never started up with anything, there's nothing for it to throw out. So out of this thing, I get the wrong answer.

How can you fix that? Well, there are ways to fix that. So you might say, well, that's sort of stupid. Why are you just doing all your NOT stuff at the beginning? The right way to implement NOT is to realize that when you have conditions like NOT, you should generate all your answers first, and then with each of these dictionaries pass along until at the very end I'll do filtering. And there are implementations of logic languages that work like that that solve this particular problem.

However, there's a more profound problem, which is which one of these is the right answer? Is it Mount Olympus or is it nothing? So you might say it's Mount Olympus, because after all, Zeus is in that database, and Zeus was neither mortal nor fallible. So you might say Zeus wants to satisfy NOT mortal Zeus or NOT fallible Zeus. But let's actually look at that database. Let's look at it.

There's no way-- how does it know that Zeus is not fallible? There's nothing in there about that. What's in there is that humans are fallible. How does it know that Zeus is not mortal? There's nothing in there about that. It just said I don't have any rule, which-- the only way I can deduce something's mortal is if it's human, and that's all it really knows about mortal. And in fact, if you remember your classical mythology, you know that the Greek gods were not mortal but fallible. So the answer is not in the rules there.

See, why does it deduce that? See, Socrates would certainly not have made this error of logic. What NOT needs in this language is not NOT. It's not the NOT of logic. What NOT needs in this language is not deducible from things in the database as opposed to not true. That's a very big difference. Subtle, but big.

So, in fact, this is perfectly happy to say not anything that it doesn't know about. So if you ask it is it not true that Zeus likes chocolate ice cream? It will say sure, it's not true. Or anything else or anything it doesn't know about. NOT means not deducible from the things you've told me. In a world where you're identifying not deducible with, in fact, not true, this is called the closed world assumption.

The closed world assumption. Anything that I cannot deduce from what I know is not true, right? If I don't know anything about x, the x isn't true. That's very dangerous. From a logical point of view, first of all, it doesn't really makes sense. Because if I don't know anything about x, I'm willing to say not x. But am I willing to say not not x? Well, sure, I don't know anything about that either maybe. So not not x is not necessarily the same as x and so on and so on and so on, so there's some sort of funny bias in there. So that's sort of funny.

The second thing, if you start building up real reasoning programs based on this, think how dangerous that is. You're saying I know I'm in a position to deduce everything true that's relevant to this problem. I'm reasoning, and built into my reasoning mechanism is the assumption that anything that I don't know can't possibly be relevant to this problem, right?

There are a lot of big organizations that work like that, right? Most corporate marketing divisions work like that. You know the consequences to that. So it's very dangerous to start really typing in these big logical implication systems and going on what they say, because they have this really limiting assumption built in. So you have to be very, very careful about that. And that's a deep problem. That's not a problem about we can make a little bit cleverer implementation and do the filters and organize the infinite loops to make them go away. It's a different kind of problem. It's a different semantics.

So I think to wrap this up, it's fair to say that logic programming I think is a terrifically exciting idea, the idea that you can bridge this gap from the imperative to the declarative, that you can start talking about relations and really get tremendous power by going above the abstraction of what's my input and what's my output. And linked to logic, the problem is it's a goal that I think has yet to be realized.

And probably one of the very most interesting research questions going on now in languages is how do you somehow make a real logic language? And secondly, how do you bridge the gap from this world of logic and relations to the worlds of more traditional languages and somehow combine the power of both. OK, let's break.

AUDIENCE: Couldn't you solve that last problem by having the extra rules that imply it? The problem here is you have the definition of something, but you don't have the definition of its opposite. If you include in the database something that says something implies mortal x, something else implies not mortal x, haven't you basically solved the problem?

PROFESSOR: But the issue is do you put a finite number of those in?

AUDIENCE: If things are specified always in pairs--

PROFESSOR: But the impression is then what do you do about deduction? You can't specify NOTs. But the problem is, in a big system, it turns out that might not be a finite number of things. There are also sort of two issues. Partly it might not be finite. Partly it might be that's not what you want.

So a good example would be suppose I want to do connectivity. I want a reason about connectivity. And I'm going to tell you there's four things: a and b and c and d. And I'll tell you a is connected to b and c's connected to d. And now I'll tell you is a connected to d? That's the question.

There's an example where I would like something like the closed world assumption. That's a tiny toy, but a lot of times, I want to be able to say something like anything that I haven't told you, assume is not true. So it's not as simple as you only want to put in explicit NOTs all over the place. It's that sometimes it really isn't clear what you even want. That having to specify both everything and not everything is too precise, and then you get down into problems there. But there are a lot of approaches that explicitly put in NOTs and reason based on that. So it's a very good idea. It's just that then it starts becoming a little cumbersome in the very large problems you'd like to use.

AUDIENCE: I'm not sure how directly related to the argument this is, but one of your points was that one of the dangers of the closed rule is you never really know all the things that are there. You never really know all the parts to it. Isn't that a major problem with any programming? I always write programs where I assume that I've got all the cases, and so I check for them all or whatever, and somewhere down the road, I find out that I didn't check for one of them.

PROFESSOR: Well, sure, it's true. But the problem here is it's that assumption which is the thing that you're making if you believe you're identifying this with logic. So you're quite right. It's a situation you're never in. The

problem is if you're starting to believe that what this is doing is logic and you look at the rules you write down and say what can I deduce from them, you have to be very careful to remember that NOT means something else. And it means something else based on an assumption which is probably not true.

AUDIENCE: Do I understand you correctly that you cannot fix this problem without killing off all possibilities of inference through altering NOT?

PROFESSOR: No, that's not quite right. There are other-- there are ways to do logic with real NOTs. There are actually ways to do that. But they're very inefficient as far as anybody knows. And they're much more-- the, quote, inference in here is built into this unifier and this pattern matching unification algorithm. There are ways to automate real logical reasoning. But it's not based on that, and logic programming languages don't tend to do that because it's very inefficient as far as anybody knows. All right, thank you.