Slide 6.1.1
We introduced first-order resolution and said there were two issues to resolve before we could do it. First, was conversion to clausal form, which we've done. Now we have to figure out how to instantiate the variables in the universal statements. In this problem, it was clear that \( A \) was the relevant individual. But it is not necessarily clear at all how to do that automatically.

Slide 6.1.2
In order to derive an algorithmic way of finding the right instantiations for the universal variables, we need something called substitutions.

Slide 6.1.3
Here's an example of what we called an atomic sentence before, a predicate applied to some terms. There are two variables here: \( x \) and \( y \). We can think of many different ways to substitute terms into this expression. Those are called substitution instances of the expression.

Slide 6.1.4
A substitution is a set of variable-term pairs, written this way. It says that whenever you see variable \( \psi_i \), you should substitute in term \( t_j \). There should not be more than one entry for a single variable.
Slide 6.1.5
So here's one substitution instance: \( P(x, f(w), B) \). It's not particularly interesting. It's called an alphabetic variant, because we've just substituted some different variables in for \( x \) and \( y \). In particular, we've put \( z \) in for \( x \) and \( w \) in for \( y \), as shown in the substitution.

Slide 6.1.6
Here's another substitution instance of our sentence: \( P(x, f(A), B) \). We've put the constant \( A \) in for the variable \( y \).

Slide 6.1.7
To get \( P(g(z), f(A), B) \), we substitute the term \( f(z) \) in for \( x \) and the constant \( A \) for \( y \).

Slide 6.1.8
Here's one more -- \( P(C, f(A), B) \). It's sort of interesting, because it doesn't have any variables in it. We'll call an atomic sentence with no variables a ground instance. Ground means it doesn't have any variables.
You can think about substitution instances, in general, as being more specific than the original sentence. A constant is more specific than a variable. There are fewer interpretations under which a sentence with a constant is true. And even \( f(x) \) is more specific than \( y \), because the range of \( f \) might be smaller than \( U \).

You're not allowed to substitute anything in for a constant, or for a compound term (the application of a function symbol to some terms). You are allowed to substitute for a variable inside a compound term, though, as we have done with \( f \) in this example.

We'll use the notation of an expression followed by a substitution to mean the expression that we get by applying the substitution to the expression. To apply a substitution to an expression, we look to see if any of the variables in the expression have entries in the substitution. If they do, we substitute in the appropriate new expression for the variable, and continue to look for possible substitutions until no more opportunities exist.

So, in this second example, we substitute \( A \) in for \( y \), then \( y \) in for \( x \), and then we keep going and substitute \( A \) in for \( y \) again.

Now we'll look at the process of unification, which is finding a substitution that makes two expressions match each other exactly.

So, expressions \( \Omega_1 \) and \( \Omega_2 \) are unifiable if there exists a substitution \( S \) such that \( \Omega_1 S = \Omega_2 S \).
So, let's look at some unifiers of the expressions $x$ and $y$. Since $x$ and $y$ are both variables, there are lots of things you can do to make them match.

If you substitute $x$ in for $y$, then both expressions come out to be $x$.

If you put in $y$ for $x$, then they both come out to be $y$.

But you could also substitute something else, like $f(f(A))$ for $x$ and for $y$, and you'd get matching expressions.
Slide 6.1.17
Or, you could substitute some constant, like A, in for both x and y.
Some of these unifiers seem a bit arbitrary. Binding both x and y to A, or to \( f(f(A)) \) is a kind of over-commitment.

Slide 6.1.18
So, in fact, what we’re really going to be looking for is not just any unifier of two expressions, but a most general unifier, or MGU.

Slide 6.1.19
G is a most general unifier of \( \omega_1 \) and \( \omega_2 \) if and only if for all unifiers S, there exists an S-prime such that the result of applying G followed by S-prime to \( \omega_1 \) is the same as the result of applying S to \( \omega_1 \); and the result of applying G followed by S-prime to \( \omega_2 \) is the same as the result of applying S to \( \omega_2 \).

A unifier is most general if every single one of the other unifiers can be expressed as an extra substitution added onto the most general one. An MGU is a substitution that you can make that makes the fewest commitments, and can still make these two expressions equal.

Slide 6.1.20
So, let’s do a few examples together. What’s a most general unifier of \( P(x) \) and \( P(A) \)? A for x.
Slide 6.1.21
What about these two expressions? We can make them match up either by substituting x for y, or y for x. It doesn't matter which one we do. They're both “most general”.

Slide 6.1.22
Okay. What about this one? It's a bit tricky. You can kind of see that, ultimately, all of the variables are going to have to be the same. Matching the arguments to g forces y and x to be the same. And since z and y have to be the same as well (to make the middle argument match), they all have to be the same variable. Might as well make it x (though it could be any other variable).

Slide 6.1.23
What about P(x, B, B) and P(A, y, z)? It seems pretty clear that we're going to have to substitute A for x, B for y, and B for Z.

Slide 6.1.24
Here's a tricky one. It looks like x is going to have to simultaneously be g(f(v)) and g(u). How can we make that work? By substituting f(v) in for u.
Now, let’s try unifying \( P(x, f(x)) \) with \( P(x, x) \). The temptation is to say \( x \) has to be \( f(x) \), but then that \( x \) has to be \( f(x) \), etc. The answer is that these expressions are not unifiable.

The last time I explained this to a class, someone asked me what would happen if \( f \) were the identity function. Then, couldn’t we unify these two expressions? That’s a great question, and it illustrates a point I should have made before. In unification, we are interested in ways of making expressions equivalent, in every interpretation of the constant and function symbols. So, although it might be possible for the constants \( A \) and \( B \) to be equal because they both denote the same object in some interpretation, we can’t unify them, because they aren’t required to be the same in every interpretation.

An MGU can be computed recursively, given two expressions \( x, y \), to be unified, and a substitution that contains substitutions that must already be made. The argument \( s \) will be empty in a top-level call to unify two expressions.

The algorithm returns a substitution if \( x \) and \( y \) are unifiable in the context of \( s \), and fail otherwise. If \( s \) is already a failure, we return failure.

If \( x \) is equal to \( y \), then we don’t have to do any work and we return \( s \), the substitution we were given.
Slide 6.1.29
If either \( x \) or \( y \) is a variable, then we go to a special subroutine that's shown on the next slide.

Unification Algorithm

\[
\text{unify}(\text{Expr } x, \text{Expr } y, \text{Subst } s)\
\begin{align*}
\text{if } s &= \text{fail}, \text{return } \text{fail} \\
\text{else if } x &= y, \text{return } s \\
\text{else if } x &= \text{variable}, \text{return } \text{unify-\text{var}(x, y, s)} \\
\text{else if } y &= \text{variable}, \text{return } \text{unify-\text{var}(x, y, s)} \\
\text{else if } x &= \text{predicate or function application,} \\
\text{if } y \text{ has the same operator,} \\
\text{return } \text{unify-\text{arity}(\text{args}(x), \text{args}(y), s)} \\
\text{else return } \text{fail}
\end{align*}
\]

Slide 6.1.30
If \( x \) is a predicate or a function application, then \( y \) must be one also, with the same predicate or function.

Unification Algorithm

\[
\text{unify}(\text{Expr } x, \text{Expr } y, \text{Subst } s)\
\begin{align*}
\text{if } s &= \text{fail}, \text{return } \text{fail} \\
\text{else if } x &= y, \text{return } s \\
\text{else if } x &= \text{variable}, \text{return } \text{unify-\text{var}(x, y, s)} \\
\text{else if } y &= \text{variable}, \text{return } \text{unify-\text{var}(x, y, s)} \\
\text{else if } x &= \text{predicate or function application,} \\
\text{if } y \text{ has the same operator,} \\
\text{return } \text{unify-\text{arity}(\text{args}(x), \text{args}(y), s)} \\
\text{else return } \text{fail}
\end{align*}
\]

Slide 6.1.31
If so, we'll unify the lists of arguments from \( x \) and \( y \) in the context of \( s \).

Unification Algorithm

\[
\text{unify}(\text{Expr } x, \text{Expr } y, \text{Subst } s)\
\begin{align*}
\text{if } s &= \text{fail}, \text{return } \text{fail} \\
\text{else if } x &= y, \text{return } s \\
\text{else if } x &= \text{variable}, \text{return } \text{unify-\text{var}(x, y, s)} \\
\text{else if } y &= \text{variable}, \text{return } \text{unify-\text{var}(x, y, s)} \\
\text{else if } x &= \text{predicate or function application,} \\
\text{if } y \text{ has the same operator,} \\
\text{return } \text{unify-\text{arity}(\text{args}(x), \text{args}(y), s)} \\
\text{else return } \text{fail}
\end{align*}
\]

Slide 6.1.32
If not, that is, if \( x \) and \( y \) have different predicate or function symbols, we simply fail.
Finally, (if we get to this case, then \( x \) and \( y \) have to be lists, or something malformed), we go down the lists, unifying the first elements, then the second elements, and so on. Each time we unify a pair of elements, we get a new substitution that records the commitments we had to make to get that pair of expressions to unify. Each further unification must take place in the context of the commitments generated by the previous elements of the lists.

Because, at each stage, we find the most general unifier, we make as few commitments as possible as we go along, and therefore we never have to back up and try a different substitution.

**Unification Algorithm**

```plaintext
unify(Expr x, Expr y, Subst s){
    if x = fail, return fail
    else if x = y, return s
    else if x is a variable, return unify-var(x, y, s)
    else if y is a variable, return unify-var(y, x, s)
    else if x is a predicate or function application, if y has the same operator,
        return unify(args(x), args(y), s)
    else return fail
    ; x and y have to be lists
    return unify(rest(x), rest(y),
        unify(first(x), first(y), s))
}
```

Slide 6.1.34
Given a variable \( \text{var} \), an expression \( \text{x} \), and a substitution \( \text{s} \), we need to return a substitution that unifies \( \text{var} \) and \( \text{x} \) in the context of \( \text{s} \). What makes this tricky is that we have to first keep applying the existing substitutions in \( \text{s} \) to \( \text{var} \), and to \( \text{x} \), if it is a variable, before we’re down to a new concrete problem to solve.

Slide 6.1.35
So, if \( \text{var} \) is bound to \( \text{val} \) in \( \text{s} \), then we unify that value with \( \text{x} \), in the context of \( \text{s} \) (because we’re already committed that \( \text{val} \) has to be substituted for \( \text{var} \)).

Slide 6.1.36
Similarly, if \( \text{x} \) is a variable, and it is bound to \( \text{val} \) in \( \text{s} \), then we have to unify \( \text{var} \) with \( \text{val} \) in \( \text{s} \). (We call `unify-var` directly, because we know that \( \text{var} \) is still a `var`).
6.034 Notes: Section 6.2

Now we know how to convert to clausal form and how to do unification. So now it's time to put it all together into first-order resolution.
Here's the rule for first-order resolution. It says if you have a formula alpha or phi and another formula not psi or beta, and you can unify phi and psi with unifier theta, then you're allowed to conclude alpha or beta with the substitution theta applied to it.

Let's look at an example. Let's say we have P(x) or Q(x,y) and we also have not P(A) or R(B,z). What are we going to be able to resolve here? We look for two literals that are negations of one another, and try to resolve them. It looks like we can resolve P(x) and not P(A), so P(x) will be phi, Q(x,y) will be alpha, P(A) will be psi and R(B,z) will be beta. The unifier will be \{x/A\}.

So, we get rid of the P literals, and end up with Q(x,y) v R(B,z), but then we have to apply our substitution (the most general unifier that was necessary to make the literals match) to the result.

Finally, we end up with Q(A,y) or R(B,z).
Now let's explore what happens if we have X's in the other formula. What if we replaced the z in the second sentence by an x?

The x's in the two sentences are actually different. There is an implicit universal quantifier on the outside of each of these sentences (remember that during the process of conversion to clausal form, we first get rid of the existentially quantified variables, then drop the remaining quantifiers, which are over universally quantified variables.) So, in order to avoid being confused by the fact that these two variables named x need not refer to the same thing, we will "rename them apart".

So that means that before you try to do a resolution step, you're really supposed to rename the variables in the two sentences so that they don't share any variables in common. You won't usually need to do this that explicitly on your paper as you work through a proof, but if you were going to implement resolution in a computer program, or if you find yourself with the same variable in both sentences and it's getting confusing, then you should rename the sentences apart.

The easiest thing to do is to just go through and give every variable a new name. It's OK to do that. You just have to do it consistently for each clause. So you could rename to P of X1 or Q of X1, Y1, and you can name this one not P of A or R of B, X2. And then you could apply the resolution rule and you don't get into any trouble.

Okay. Now that we know how to do resolution, let's practice it on the example that we started in the section on clausal form. We want to prove that curiosity killed the cat.
Here are the clauses that we got from the original axioms.

Now we assert the negation of the thing we're trying to prove, so we have not K(C,T).

We can apply the resolution rule to any pair of lines that contain unifiable literals. Here's one way to do the proof. We'll use the "set-of-support" heuristic (which says we should involve the negation of the conclusion in the proof), and resolve away K(C,T) from lines 5 and 8, yielding K(J,T).

Then, we can resolve C(T) and C(x) in lines 6 and 7 by substituting T for x, and getting A(T).
Using lines 4 and 9, and substituting J for x and T for y, we get not L(J) or not A(T).

From lines 10 and 11, we get not L(J).

From 3 and 12, substituting J for x, we get not D(y) or not O(J,y).

From 13 and 2, substituting Fido for x, we get not D(Fido).
Slide 6.2.18
And finally, from lines 14 and 1, we derive a contradiction. Yay! Curiosity did kill the cat.

Slide 6.2.19
So, if we want to use resolution refutation to prove that something is valid, what would we do? What do we normally do when we do a proof using resolution refutation?

Slide 6.2.20
We say, well, if I know all these things, I can prove this other thing I want to prove. We prove that the premises entail the conclusion.

Slide 6.2.21
What does it mean for a sentence to be valid, in the language of entailment? That it's true in all interpretations. What that means really is that it should be derivable from nothing. A valid sentence is entailed by the empty set of sentences. The valid sentence is true no matter what. So we're going to prove something with no assumptions.
We can prove it by resolution refutation by negating the sentence and trying to derive a contradiction.

So, let's do an example. Imagine that we would like to show the validity of this sentence, which is a classical Aristotelian syllogism.

We start by negating it and converting to clausal form. We get rid of the arrows and drive in negations to arrive at this sentence in clausal form.

We enter the clauses into our proof.
6.034 Notes: Section 6.3

Slide 6.3.1
In this section, we're going to look at three techniques for making logical proof more useful, and then conclude by talking about the limits of first-order logic.

Slide 6.3.2
The version of the first-order resolution rule that we have shown you is called binary resolution because it involves two literals, one from each clause being resolved. It turns out that this form of resolution is not complete. There are sets of unsatisfiable clauses that will not generate a contradiction by successive applications of binary resolution.

Slide 6.3.3
Here's a pair of clauses. P(x) v P(y) and not P(v) or not P(w). Can we get a contradiction from them using binary resolution?
Slide 6.3.4
We should be able to! They are clearly unsatisfiable. There is no possible interpretation that will make both of these clauses simultaneously true, since that would require \( P(A) \) and \( \neg (P(A)) \) to be true for everything in the universe.

Slide 6.3.5
But when we apply binary resolution to these clauses, all we can get is something like \( P(x) \) or \( \neg P(w) \) (your variables may vary).

Slide 6.3.6
If we use binary resolution on this new clause with one of the parent clauses, we get back one of the parent clauses. We do not get a contradiction. So, we have shown by counterexample that binary resolution is not, in fact, a complete strategy.

Slide 6.3.7
It turns out that there is a simple extension of binary resolution that is complete. In that version, known as generalized resolution, we look for subsets of literals in one clause that can be unified with the negation of a subset of literals in the other clause. In our example from before, all the \( \neg P \) literals in one clause can be unified with the negation of the \( P \) literals in the other clause.
An alternative to using generalized resolution is to introduce a new inference rule, in addition to binary resolution, called factoring. In factoring, if you can unify two literals within a single clause, alpha and beta in this case, with unifier theta. Then you can drop one of them from the clause (it doesn't matter which one), and then apply the unifier to the whole clause.

So, for example, we can apply factoring to this sentence, by unifying \( P(x,y) \) and \( P(v,A) \).

We get \( Q(Y) \lor P(x,Y) \), and then we have to apply the substitution \( \{ x/v, y/A \} \), which yields the result \( Q(A) \lor P(v,A) \).

Note that factoring in propositional logic is just removing duplicate literals from sentences, which is obvious and something we've been doing without comment.

And binary resolution, combined with factoring is complete in a sense that we'll study more carefully later in this section.
Slide 6.3.12
One thing you can do with resolution is ask for an answer to a question. If your desired conclusion is that there exists an x such that P(x), then in the course of doing the proof, we’ll figure out what value of x makes P(x) true. We might be interested in knowing that answer. For instance, it’s possible to do a kind of planning via theorem proving, in which the desired conclusion is “There exists a sequence of action such that, if I do them in my initial state, my goal will be true at the end.” Generally, you’re not just interested in whether such a sequence exists, but in what it is.

One way to deal with this is Green’s trick, named after Cordell Green, who pioneered the use of logic in software engineering applications. We’ll see it by example.

Slide 6.3.13
Let’s say we know that all men are mortal and that Socrates is a man. We want to know whether there are any mortals. So, our desired conclusion, negated and turned into clausal form would be not mortal(x). Green’s trick will be to add a special extra literal onto that clause, of the form Answer(x).

Slide 6.3.14
Now, we do resolution as before, but when we come to a clause that contains only the answer literal, we stop. And whatever the variable x is bound to in that literal is our answer.

Slide 6.3.15
When we defined the language of first-order logic, we defined a special equality predicate. And we also defined special semantics for it (the sentence term 1 equals term 2 holds in an interpretation if and only if term1 and term2 both denote the same object in that interpretation). In order to do proofs that contain equality statements in them, we have to add a bit more mechanism.
One strategy is to add one more special proof rule, just as we did with factoring. The new proof rule is called paramodulation. It’s kind of hairy and hard to use, though, so we are not going to do it in this class (though it is implemented in most serious theorem provers).

Another strategy, which is easier to understand, and instructive, is to treat equality almost like any other predicate, but to constrain its semantics via axioms.

Just to make it clear that Equals is a predicate that we’re going to handle normally in resolution, we’ll write it with a word rather than the equals sign.

Equals has two important sets of properties. The first three say that it is an equivalence relation. First, it’s symmetric: every \( x \) is equal to itself.

Second, it’s reflexive. If \( x \) is equal to \( y \) then \( y \) is equal to \( x \).
Slide 6.3.20
Third, it's transitive. That means that if x equals y and y equals z, then x equals z.

Slide 6.3.21
The other thing we need is the ability to "substitute equals for equals" into any place in any predicate. That means that, for each place in each predicate, we'll need an axiom that looks like this: for all x and y, if x equals y, then if P holds of x, it holds of y.

Proof Example
- Let's go back to our old geometry domain and try to prove what the hat of A is.

Slide 6.3.22
Let's go back to our old geometry domain and try to prove what the hat of A is.

Proof Example
- Let's go back to our old geometry domain and try to prove what the hat of A is.
- Axioms in FOL (plus equality axioms)

\[
\begin{align*}
\text{Above}(A, C) \\
\text{Above}(B, D) \\
\neg \exists x. \text{Above}(x, A) \\
\neg \exists x. \text{Above}(x, B) \\
\forall x, y. \text{Above}(x, y) \rightarrow \text{hat}(y) = x \\
\forall x. (\neg \exists y. \text{Above}(y, x)) \rightarrow \text{hat}(x) = x
\end{align*}
\]
Let's see if we can derive that, using resolution refutation and Green's trick.

Here's the result of my clausal-form converter run on those axioms.

Now, our goal is to prove \( \exists x \) such that \( \text{Eq}(\hat{A}, x) \). That is negated and turned into clausal form, yielding \( \neg \text{Eq}(\hat{A}, x) \). And we add in the answer literal, so we can keep track of what the answer is.

Here's the proof. The answer is A! And we figured this out without any kind of enumeration of interpretations.
Slide 6.3.28
What if we wanted to use the same axioms to figure out what the hat of D is? We just change our query and do the proof. Here it is.

Slide 6.3.29
Here's a worked example of a problem with equality.

Slide 6.3.30
Now, let's see what we can say, in general, about proof in first-order logic.
Remember that a proof system is complete, if, whenever the KB entails S, we can prove S from KB.

Slide 6.3.31
In 1929, Gödel proved a completeness theorem for first-order logic: There exists a complete proof system for FOL. But, living up to his nature as a very abstract logician, he didn't come up with such a proof system; he just proved one existed.
6.3.32
Then, in 1965, Robinson came along and showed that resolution refutation is sound and complete for FOL.

6.3.33
So, we know that if a proof exists, then we can eventually find it with resolution. Unfortunately, we no longer know, as we did in propositional resolution, that eventually the process will stop. So, it's possible that there is no proof, and that the resolution process will run forever.

This makes first-order logic what is known as “semi-decidable”. If the answer is “yes”, that is, if there is a proof, then the theorem prover will eventually halt and say so. But if there isn't a proof, it might run forever!

6.3.34
So, things are relatively good with regular first-order logic. And they're still fine if you add addition to the language. But if you add addition and multiplication, it starts to get weird!

6.3.35
In 1931, Gödel proved an incompleteness theorem, which says that there is no consistent, complete proof system for FOL plus arithmetic. (Consistent is the same as sound). Either there are sentences that are true, but not provable, or there are sentences that are provable, but not true. It's not so good either way.
Slide 6.3.36
Here's the roughest cartoon of how the proof goes. Arithmetic gives you the ability to construct code names for sentences within the logic, and therefore to construct sentences that are self-referential. This sentence, \( P \), is sometimes called the Godel-sentence. \( P \) is "P is not provable".

Slide 6.3.37
If \( P \) is true, then \( P \) is not provable (so the system is incomplete). If \( P \) is false, then \( P \) is provable (so the system is inconsistent).

This result was a huge blow to the current work on the foundations of mathematics, where they were, essentially, trying to formalize all of mathematical reasoning in first-order logic. And it pre-figured, in some sense, Turing's work on uncomputability.

Ultimately, though, just as uncomputability doesn't worry people who use computer programs for practical applications, incompleteness shouldn't worry practical users of logic.

Slide 6.4.1
Now that we've studied the syntax and semantics of logic, and know something about how to do inference in it, we're going to talk about how logic has been applied in real domains, and look at an extended example.

Slide 6.4.2
There is currently a big resurgence of logical representations and inference in the context of the web. As it stands now, web pages were written in natural language (English or French, etc), by the people and for the people. But there is an increasing desire to have computer programs (web agents or 'bots) crawl the web and figure things out by "reading" web pages. As we'll see in the next module of this course, it can be quite hard to extract the meaning from text written in natural language. So the World-Wide Web Consortium, in conjunction with people in universities and industry, are defining a standard language, which is essentially first-order logic, for formally encoding information in web pages. Information that is written in this formal language will be much easier to extract automatically.
Slide 6.4.3
It is becoming more appealing, in business, to have computers talk directly to one another, and to leave humans out of the loop. One place this can happen is in negotiating simple contracts between companies to deliver goods at some price. Benjamin Grosof, who is a professor in the Sloan School, works on using non-monotonic logic (a version of first-order logic, in which you're allowed to have conflicting rules, and have a system for deciding which ones have priority) to specify a company's business rules.

Slide 6.4.4
Another example, which we'll pursue in detail, is the language the airlines use to specify the rules on their airfares. It turns out that every day, many times a day, airlines revise and publish (electronically) their fare structures. And, as many of you know, the rules governing the pricing of airplane tickets are pretty complicated, and certainly unintuitive. In fact, they're so complicated that the airlines had to develop a formal language that is similar to logic, in order to describe their different kinds of fares and the restrictions on them. Amazingly, there are on the order of 20 million different fares! To generate a price for a particular proposed itinerary, it requires piecing together a set of fares to cover the parts of the itinerary. Typically, the goal is to find the cheapest such set of fares, subject to some constraints.

Slide 6.4.5
We're not going to worry about how to do the search to find the cheapest itinerary and fare structure (that's a really hard and interesting search problem!). Instead, we'll just think about pricing a particular itinerary. Pricing an airline ticket is not as simple as adding up the prices for the individual flight legs. There are many different pricing schemes, each depending on particular attributes of the combination of flights that the passenger proposes to take. For instance, at some point in 1998, American Airlines had 29 different fares for going from Boston to San Francisco, ranging in price from $1943 to $231, each with a different constraint on its use. In this discussion, we won't get into the actual ticket prices; instead we'll work on writing down the logical expressions that describe when a particular fare applies to a proposed itinerary.

Slide 6.4.6
Here are some examples of airfare restrictions that we might want to encode logically:

- The passenger is under 2 or over 65
- The passenger is accompanying another passenger who is paying full fare
- It doesn't go through an expensive city
- There are no flights during rush hour (defined in local time)
- The itinerary stays over a Saturday night
- Layovers are legal: not too short; not too long
- Round-the-world itinerary that doesn't backtrack
- The itinerary is a regular two-trip round-trip
- This price applies to one flight, as long as there is no other flight in this itinerary operated by EL Cheapo Air
- If the sum of the fares for a circle trip with three legs is less than the "comparable" round-trip price between the origin and any stopover point, then you must add a surcharge.
Slide 6.4.7
The first step in making a logical formalization of a domain is coming up with an ontology. According to Leibniz (a philosopher from the 17th century), ontology is "the science of something and of nothing, of being and not-being, of the thing and the mode of the thing, of substance and accident." Whoa! I wish our lecture notes sounded that deep.

Slide 6.4.8

Slide 6.4.9
For us, more prosaically, an ontology will be a description of the kinds of objects that you have in your world and their possible properties and relations. Making up an ontology is a lot like deciding what classes and methods you'll need when you design an object-oriented program.

Slide 6.4.10
Okay. So what are the kinds of things we have in the airfare domain? That's a hard question, because it depends on the level of abstraction at which we want to make our model. Probably we don't want to talk about particular people or airplanes; but we might need to talk about people in general, in terms of various properties (their ages, for example, but not their marital status), or airplane types. We will need a certain amount of detail though, so, for instance, it might matter which airport within a city you're using, or which terminal within an airport. Often you have to adjust the level of abstraction that you use as you go along.
6.4.11 Here’s a list of the relevant object types I came up with:

- passenger
- flight
- city
- airport
- terminal
- flight segment (a list of flights, to be flown all in one “day”)
- itinerary (a passenger and a list of flight segments)

6.4.12 We’ll also need some non-concrete object types, including:

- list
- number

6.4.13 Once we know what kinds of things we have in our world, we need to come up with a vocabulary of constant names, predicate symbols, and function symbols that we’ll use to talk about their properties and relations.

There are two parts to this problem. We have to decide what properties and relations we want to be able to represent, and then we have to decide how to represent them.

6.4.14 Let’s talk, for a minute, about something simple, like saying that an object named P is red. There are a number of ways to say this, including:

- Red(P)
- Color(P, Red)
- Color(P) = Red
- Property(P, Color, Red)
Slide 6.4.15
Let's look at the difference between the first two. Red(P) seems like the most straightforward way to say that P is red. But what if we wanted to write a rule saying that all the blocks in a particular stack, S, are the same color? Using the second representation, we could say:

$$\exists c. \forall b. \text{In}(b, S) \rightarrow \text{Color}(b, c)$$

In this case, we have *reified* redness; that is, we've made it into an object that can be named and quantified over. It will turn out that it's often useful to use this kind of representation.

Slide 6.4.16
It's possible to go even farther down this road, in case, for instance, we wanted to say that all the blocks in stack S have all the same properties:

$$\forall p. \exists v. \forall b. \text{In}(b, S) \rightarrow \text{Property}(b, p, v)$$

That is, for every property, there's a value, such that every block in S has that value for the property. Some people advocate writing all logical specifications using this kind of formalization, because it is very general; but it's also kind of hard to read. We'll stick to representations closer to \(\text{Color}(P, \text{Red})\).

Slide 6.4.17
The particular relations we'll use come from two sources. Some relations will be used to specify the basic facts in our knowledge-base.

Slide 6.4.18
Here are some of the basic relations in our domain. I've named the arguments with the types of objects that we expect to be there. This is just an informal convention to show you how we intend to use the relations. (There are logics that are strongly typed, which require you to declare the types of the arguments to each relation or function).
So, we might describe passenger Fred using the sentences:

- \( \text{Age}(\text{Fred}, 47) \)
- \( \text{Nationality}(\text{Fred}, \text{US}) \)
- \( \neg \text{Wheelchair}(\text{Fred}) \)

This only serves to encode a very simple version of this domain. We haven't started to talk about time zones, terminals, metropolitan areas (usually it's okay to fly into San Jose and then out of San Francisco, as if they were the same city), airplane types, how many reservations a flight currently has, and so on and on.

Other relations will be used to express the things we want to infer. An example in this domain might be \( \text{Qualifies_for_fare_class_37} \) or something else equally intuitive. As we begin trying to write down a logical expression for \( \text{Qualifies_for_fare_class_37} \) in terms of the basic relations, we'll find that we want to define more intermediate relations. It will be exactly analogous to defining functions when writing a Scheme program: it's not strictly necessary, but no-one would ever be able to read your program (and you probably wouldn't be able to write it correctly) if you didn't.

We will often define relations using implications rather than equivalence. It makes it easier to add additional pieces of the definition (circumstances in which the relation would be true).

However, written this way, we can't infer anything from knowing that the relation holds of some objects. If we need to do that, we have to write out the equivalence.
Okay. Let's start with a very simple rule. Let's say that an itinerary has the *Infant_Fare* property if the passenger is under age 2. We can write that as

\[
\text{all } i, a, p. \text{Passenger}(i, p) \land \text{Age}(p, a) \land a < 2 \rightarrow \text{Infant}_\text{Fare}(i)
\]

It's not completely obvious that this is the right way to write the rule. For instance, it's useful to note that this is equivalent to saying

\[
\text{all } i. (\exists a, p. \text{Passenger}(i, p) \land \text{Age}(p, a) \land a < 2) \rightarrow \text{Infant}_\text{Fare}(i)
\]

This second form is clearer (though the first form is closer to what we'll use next, when we talk about rule-based systems). Note, also, that changing the implication to equivalence in these two statements makes them no longer be equivalent.

We just snuck something in here: \(a < 2\). We'll need some basic arithmetic in almost any interesting logic domain. How can we deal with that? We saw, in the previous section, that adding arithmetic including multiplication means that our language is no longer complete. But that's the sort of thing that worries logicians more than practitioners.
Slide 6.4.27
In this domain we'll need addition and subtraction and greater-than. One strategy would be to axiomatize them in logic, but that's usually wildly inefficient. Most systems for doing practical logical inference include built-in logical predicates that will automatically evaluate themselves if their arguments are instantiated. So if, during the course of a resolution proof, you had a clause of the form
\[ P(a) \lor (3 < 2) \lor (a > 1 + 2) \]
it would automatically simplify to
\[ P(a) \lor (a > 3) \]

Slide 6.4.28
Okay. Now, let's go to a significantly harder one. This isn't exactly a fare restriction; it's more of a correctness criterion on the flights within the itinerary. The idea is that an itinerary can be made up of multiple flight segments; each flight segment might, itself, be made up of multiple flights. In order for the itinerary to be well-formed, the flight segments are not required to have any particular relation to each other. However, there are considerable restrictions on a flight segment. One way to think about a flight segment is as a sequence of flights that you would do in one day (though it might actually last for more than one day if you're going for a long time).

Slide 6.4.29
For a flight segment to be well-formed, it has to satisfy the following properties:

- The departure and arrival airports match up correctly
- The layovers (gaps between arriving in an airport and departing from it) aren't too short (so that there's a reasonable probability that the passenger will not miss the connection
- The layovers aren't too long (so that the passenger can't spend a week enjoying him or herself in the city; we need to be sure to charge extra for that!)

So, let's work toward developing a logical specification of the well-formedness of a flight segment. A flight segment is a list of flights. So, we'll make a short detour to talk about lists in logic, then come back to well-formed flight-segments.

Slide 6.4.30
In the list of relations for the domain, we included a constant Nil and a function cons, without explanation. Here's the explanation.
Slide 6.4.31
We can make and use lists in logic, much as we might do in Scheme. We have a constant that stands for the empty list. Then, we have a function cons that, given any object and a list, denotes the list that has the object argument as its head (car) and the list argument as its tail (cdr).
So cons(A,cons(B,Nil)) is a list with two elements, A and B.

Slide 6.4.32
We can also use the power of unification to specify conditions to assertions. So we can write forall x lengthOne(cons(x,Nil)), which is a more compact way of saying that every list that is equal to the cons of an element onto Nil has the property of being lengthOne.

Slide 6.4.33
Now that we know how to do some things with lists, we'll go back to the problem of ensuring that a flight segment is well-formed. This is basically a condition on all the layovers in the segment, so we'll have to run down the list making sure it's all okay.

Slide 6.4.34
We can start with a simple case. If the flight segment has one flights, then it's well-formed. We can write this as all f. WellFormed(cons(f, Nil))
Slide 6.4.35
Now, let's do the hard case. We can say that a flight segment with more than one flight is well-formed if the first two flights are contiguous (end and start in the same airport), the layover time between the first two flights is legal, and the rest of the flight segment is well-formed.

In logic, that becomes: all f1, f2, r. Contiguous(f1, f2) \land LegalLayover(f1, f2) \land WellFormed(cons(f2, r)) \rightarrow WellFormed(cons(f1, cons(f2, r)))

Slide 6.4.37
Note that we've invented some vocabulary here. Contiguous and LegalLayover are neither given to us as basic relations, nor the relation we are trying to define. We made them up, just as you make up function names, in order to divide our problem into conquerable sub-parts.

Slide 6.4.38
What makes two flights contiguous? The arrival airport of the first has to be the same as the departure airport of the second. We can write this as

all f1, f2. (exists c. Destination(f1, c) \land Origin(f2, c)) 
\rightarrow Contiguous(f1, f2)
Now, what makes layovers legal? They have to be not too short and not too long.

\[
\text{all } f_1, f_2. \quad \text{LayoverNotTooShort}(f_1, f_2) \land \text{LayoverNotTooLong}(f_1, f_2) \rightarrow \text{LayoverLegal}(f_1, f_2)
\]

Let's say that passengers need at least 30 minutes to change planes. That comes out fairly straightforwardly as

\[
\text{all } f_1, f_2. \quad (\exists \, t_1, t_2. \quad \text{ArrivalTime}(f_1, t_1) \land \text{DepartureTime}(f_2, t_2) \land (t_2 - t_1 > 30)) \rightarrow \text{LayoverNotTooShort}(f_1, f_2)
\]

This is a very simple version of the problem. You could imagine making this incredibly complex and nuanced. How long does it take someone to change planes? It might depend on: whether they're in a wheelchair, whether they have to change terminals, how busy the terminals are, how effective the inter-terminal transportation is, whether they have small children, whether the airport has signs in their native language, whether there's bad weather, how long the lines are at security, whether they're from a country whose citizens take a long time to clear immigration.

You probably wouldn't want to add each of these things as a condition in the rule about layovers. Rather, you would want this system, ultimately, to be connected to a knowledge-based of common sense facts and relationships, which could be used to deduce an expected time to make the connection. Common-sense reasoning is a fascinating area of AI with a long history. It seems to be (like many things!) both very important and very hard.

We'll continue in our more circumscribed setting, to address the question of what makes a layover not be too long. This will have an easy case and a hard case. The easy case is that a layover is not too long if it’s less than three hours:

\[
\text{all } f_1, f_2. \quad (\exists \, t_1, t_2. \quad \text{ArrivalTime}(f_1, t_1) \land \text{DepartureTime}(f_2, t_2) \land (t_2 - t_1 < 180)) \rightarrow \text{LayoverNotTooLong}(f_1, f_2)
\]
Now, for the hard case. Let's imagine you've just flown into Ulan Bator, trying to get to Paris. And there's only one flight per day from Ulan Bator. We might want to say that your layover is not too long if there are no flights from here to your next destination that go before the one you're scheduled to take (and that have an adequately long layover).

\[
\text{all } f_1, f_2 \ (\exists o, d, t_2.
\text{Origin}(f_2, o) \land \text{Destination}(f_2, d) \land \text{DepartureTime}(f_2, t_2) \land \\
\neg \exists f_3, t_3. \ (\text{Origin}(f_3, o) \land \text{Destination}(f_3, d) \land \\
\text{DepartureTime}(f_3, t_3) \land (t_3 < t_2) \\
\land \text{LayoverNotTooShort}(f_1, f_3)) \\ \rightarrow \\
\text{LayoverNotTooLong}(f_1, f_2)
\]

Of course, you can imagine all sorts of common-sense information that might influence this definition of LayoverNotTooLong, just as in the previous case.

We haven't been writing these definitions with efficiency in mind. In all likelihood, if we tried to put them into a regular theorem prover, we would never get an answer out. In the next segment of material, we'll see how to use a restricted version of first-order logic to get fairly efficient logical programs. And we'll continue this example there.