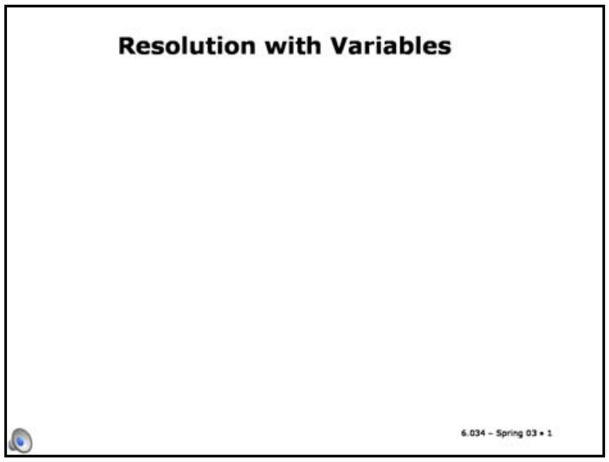
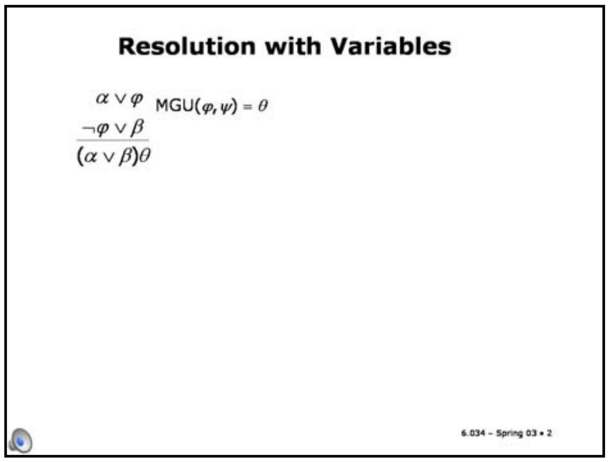
6.034 Notes: Section 10.5

Slide 10.5.1
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.



Resolution with Variables

6.034 – Spring 03 • 1



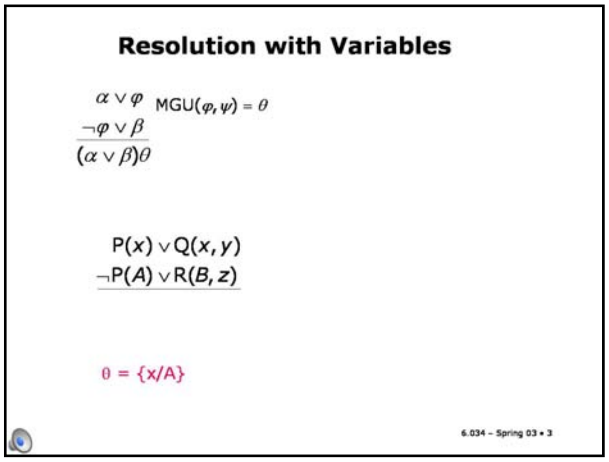
Resolution with Variables

$$\frac{\begin{array}{l}\alpha \vee \varphi \\ \neg\varphi \vee \beta\end{array}}{(\alpha \vee \beta)\theta} \quad \mathrm{MGU}(\varphi, \psi) = \theta$$

6.034 – Spring 03 • 2

Slide 10.5.2
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.

Slide 10.5.3
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}.



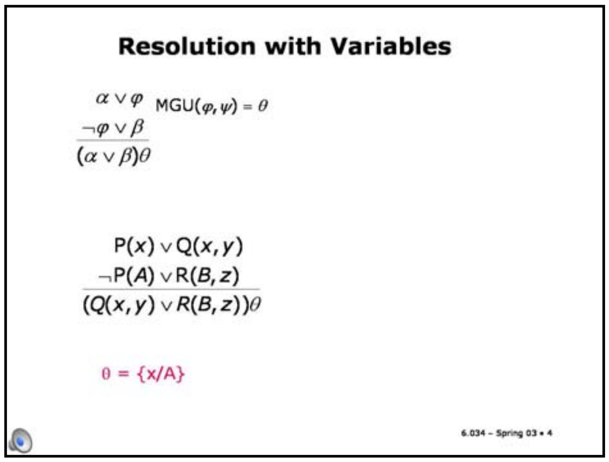
Resolution with Variables

$$\frac{\begin{array}{l}\alpha \vee \varphi \\ \neg\varphi \vee \beta\end{array}}{(\alpha \vee \beta)\theta} \quad \mathrm{MGU}(\varphi, \psi) = \theta$$

$$\frac{\begin{array}{l}\mathrm{P}(x) \vee \mathrm{Q}(x, y) \\ \neg\mathrm{P}(A) \vee \mathrm{R}(B, z)\end{array}}{}$$

$\theta = \{x/A\}$

6.034 – Spring 03 • 3



Resolution with Variables

$$\frac{\alpha \vee \varphi \qquad \text{MGU}(\varphi,\psi)=\theta \\ \neg\varphi \vee \beta}{(\alpha \vee \beta)\theta}$$

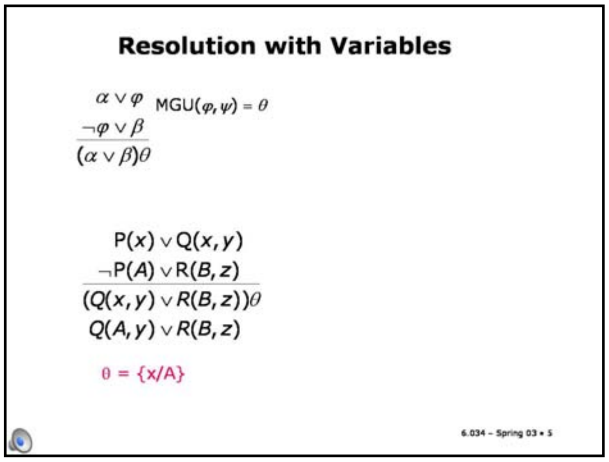
$$\frac{P(x) \vee Q(x,y) \\ \neg P(A) \vee R(B,z)}{(Q(x,y) \vee R(B,z))\theta}$$

$\theta = \{x/A\}$

6.034 – Spring 03 • 4

Slide 10.5.4
So, we get rid of the P literals, and end up with **Q(x,y) or 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.

Slide 10.5.5
Finally, we end up with **Q(A,y) or R(B,z)**.



Resolution with Variables

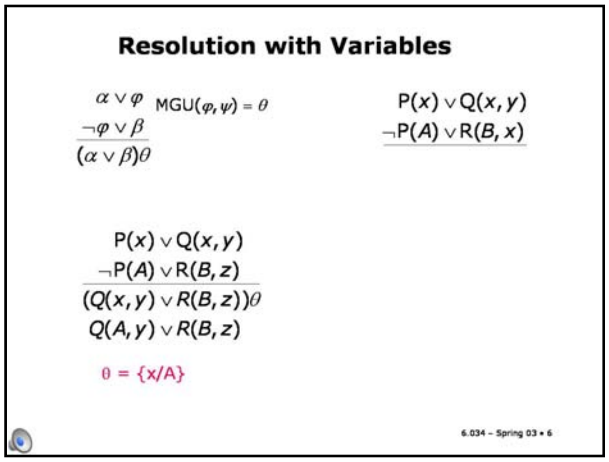
$$\frac{\alpha \vee \varphi \qquad \text{MGU}(\varphi,\psi)=\theta \\ \neg\varphi \vee \beta}{(\alpha \vee \beta)\theta}$$

$$\frac{P(x) \vee Q(x,y) \\ \neg P(A) \vee R(B,z)}{(Q(x,y) \vee R(B,z))\theta}$$

$Q(A,y) \vee R(B,z)$

$\theta = \{x/A\}$

6.034 – Spring 03 • 5



Resolution with Variables

$$\frac{\alpha \vee \varphi \qquad \text{MGU}(\varphi,\psi)=\theta \\ \neg\varphi \vee \beta}{(\alpha \vee \beta)\theta}$$

$$\frac{P(x) \vee Q(x,y) \\ \neg P(A) \vee R(B,x)}{}$$

$$\frac{P(x) \vee Q(x,y) \\ \neg P(A) \vee R(B,z)}{(Q(x,y) \vee R(B,z))\theta}$$

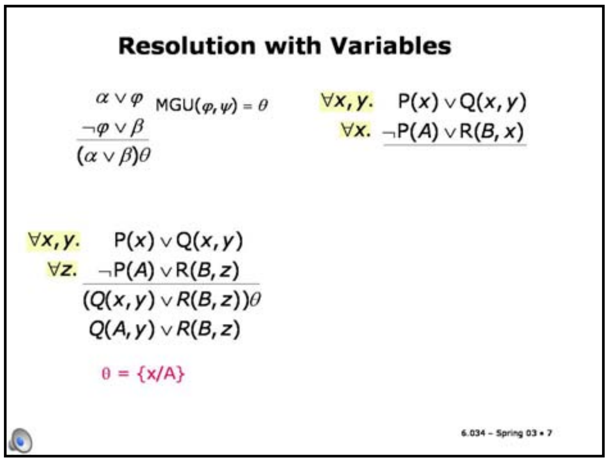
$Q(A,y) \vee R(B,z)$

$\theta = \{x/A\}$

6.034 – Spring 03 • 6

Slide 10.5.6
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?

Slide 10.5.7
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".



Resolution with Variables

$$\frac{\alpha \vee \varphi \qquad \text{MGU}(\varphi,\psi)=\theta \\ \neg\varphi \vee \beta}{(\alpha \vee \beta)\theta}$$

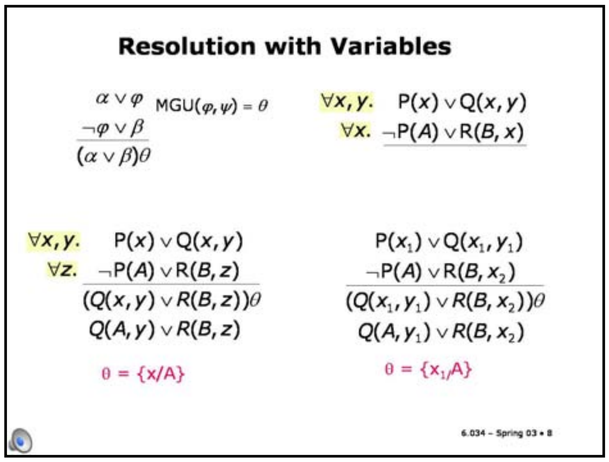
$$\frac{\forall x,y.\ P(x) \vee Q(x,y) \\ \forall x.\ \neg P(A) \vee R(B,x)}{}$$

$$\frac{\forall x,y.\ P(x) \vee Q(x,y) \\ \forall z.\ \neg P(A) \vee R(B,z)}{(Q(x,y) \vee R(B,z))\theta}$$

$Q(A,y) \vee R(B,z)$

$\theta = \{x/A\}$

6.034 – Spring 03 • 7



Resolution with Variables

$$\frac{\begin{array}{c}\alpha \vee \varphi \\ \neg\varphi \vee \beta\end{array}}{(\alpha \vee \beta)\theta} \quad MGU(\varphi,\psi)=\theta$$

$$\begin{array}{rl} \forall x,y. & P(x) \vee Q(x,y) \\ \forall x. & \neg P(A) \vee R(B,x) \end{array}$$

$$\begin{array}{rl} \forall x,y. & P(x) \vee Q(x,y) \\ \forall z. & \neg P(A) \vee R(B,z) \\ \hline & (Q(x,y) \vee R(B,z))\theta \\ & Q(A,y) \vee R(B,z) \end{array}$$

$\theta = \{x/A\}$

$$\begin{array}{l} P(x_1) \vee Q(x_1,y_1) \\ \neg P(A) \vee R(B,x_2) \\ \hline (Q(x_1,y_1) \vee R(B,x_2))\theta \\ Q(A,y_1) \vee R(B,x_2) \end{array}$$

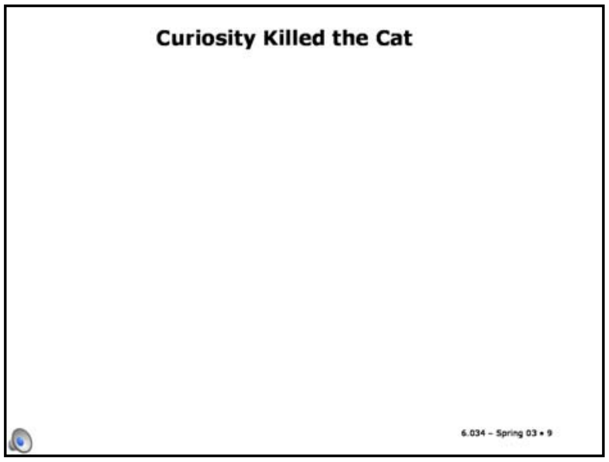
$\theta = \{x_1/A\}$

6.034 – Spring 03 • 8

Slide 10.5.8
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.

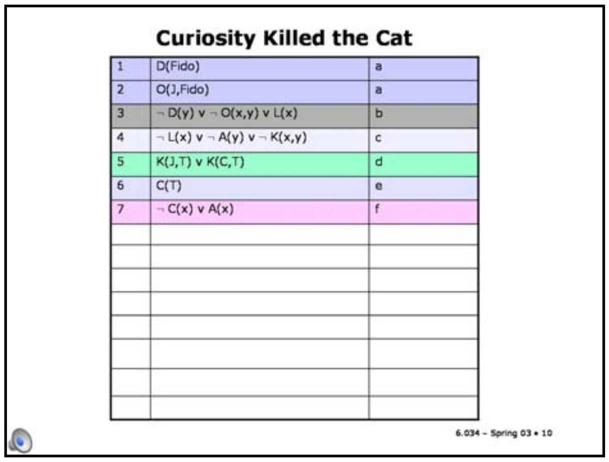
The easiest thing to do is to just go through and give every variable a new name. It's OK to do that. You just have to do it consistently for each clause. So you could rename to **P(x_1) or Q(x_1, y_1)**, and you can name this one **not P(A) or R(B, x_2)**. And then you could apply the resolution rule and you don't get into any trouble.

Slide 10.5.9
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.



Curiosity Killed the Cat

6.034 – Spring 03 • 9



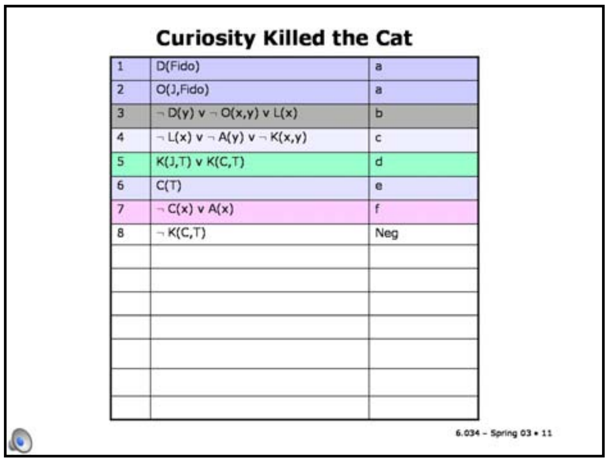
Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	¬ D(y) v ¬ O(x,y) v L(x)	b
4	¬ L(x) v ¬ A(y) v ¬ K(x,y)	c
5	K(J,T) v K(C,T)	d
6	C(T)	e
7	¬ C(x) v A(x)	f

6.034 – Spring 03 • 10

Slide 10.5.10
Here are the clauses that we got from the original axioms.

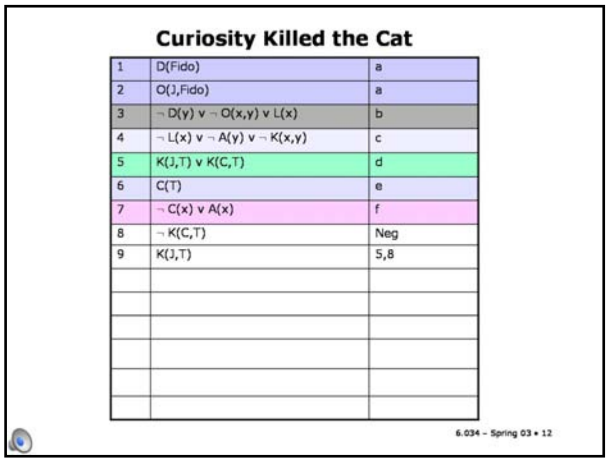
Slide 10.5.11
Now we assert the negation of the thing we're trying to prove, so we have **not K(C,T)**.



Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	¬ D(y) v ¬ O(x,y) v L(x)	b
4	¬ L(x) v ¬ A(y) v ¬ K(x,y)	c
5	K(J,T) v K(C,T)	d
6	C(T)	e
7	¬ C(x) v A(x)	f
8	¬ K(C,T)	Neg

6.034 – Spring 03 • 11



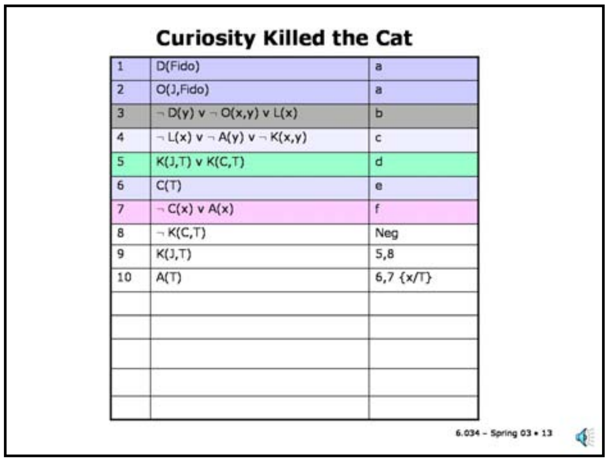
Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	¬ D(y) v ¬ O(x,y) v L(x)	b
4	¬ L(x) v ¬ A(y) v ¬ K(x,y)	c
5	K(J,T) v K(C,T)	d
6	C(T)	e
7	¬ C(x) v A(x)	f
8	¬ K(C,T)	Neg
9	K(J,T)	5,8

6.034 – Spring 03 • 12

Slide 10.5.12
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).

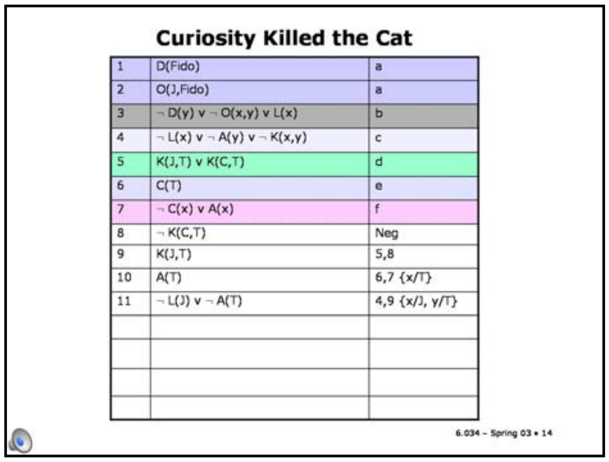
Slide 10.5.13
Then, we can resolve **C(T)** and **not C(x)** in lines 6 and 7 by substituting T for x, and getting **A(T)**.



Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	¬ D(y) v ¬ O(x,y) v L(x)	b
4	¬ L(x) v ¬ A(y) v ¬ K(x,y)	c
5	K(J,T) v K(C,T)	d
6	C(T)	e
7	¬ C(x) v A(x)	f
8	¬ K(C,T)	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}

6.034 – Spring 03 • 13



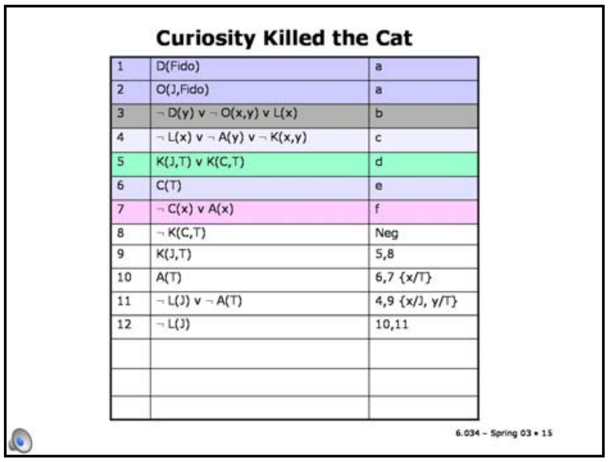
Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	¬ D(y) v ¬ O(x,y) v L(x)	b
4	¬ L(x) v ¬ A(y) v ¬ K(x,y)	c
5	K(J,T) v K(C,T)	d
6	C(T)	e
7	¬ C(x) v A(x)	f
8	¬ K(C,T)	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}
11	¬ L(J) v ¬ A(T)	4,9 {x/J, y/T}

6.034 – Spring 03 • 14

Slide 10.5.14
Using lines 4 and 9, and substituting J for x and T for y, we get not **L(J) or not A(T)**.

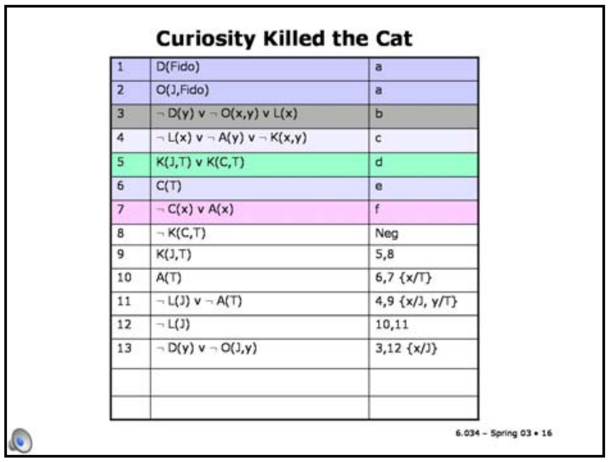
Slide 10.5.15
From lines 10 and 11, we get **not L(J)**.



Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	¬ D(y) v ¬ O(x,y) v L(x)	b
4	¬ L(x) v ¬ A(y) v ¬ K(x,y)	c
5	K(J,T) v K(C,T)	d
6	C(T)	e
7	¬ C(x) v A(x)	f
8	¬ K(C,T)	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}
11	¬ L(J) v ¬ A(T)	4,9 {x/J, y/T}
12	¬ L(J)	10,11

6.034 – Spring 03 • 15



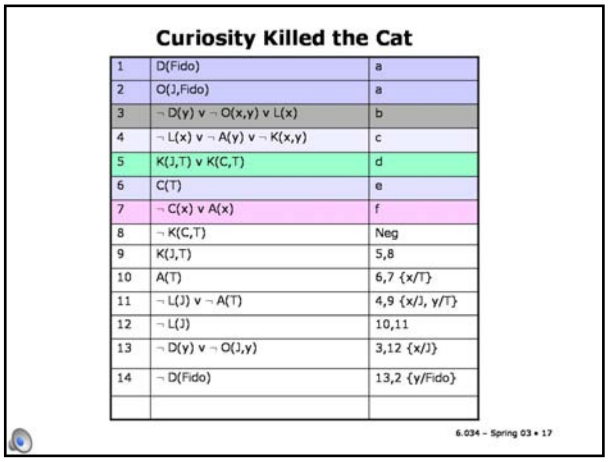
Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	¬ D(y) v ¬ O(x,y) v L(x)	b
4	¬ L(x) v ¬ A(y) v ¬ K(x,y)	c
5	K(J,T) v K(C,T)	d
6	C(T)	e
7	¬ C(x) v A(x)	f
8	¬ K(C,T)	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}
11	¬ L(J) v ¬ A(T)	4,9 {x/J, y/T}
12	¬ L(J)	10,11
13	¬ D(y) v ¬ O(J,y)	3,12 {x/J}

6.034 – Spring 03 • 16

Slide 10.5.16
From 3 and 12, substituting J for x, we get **not D(y) or not O(J,y)**.

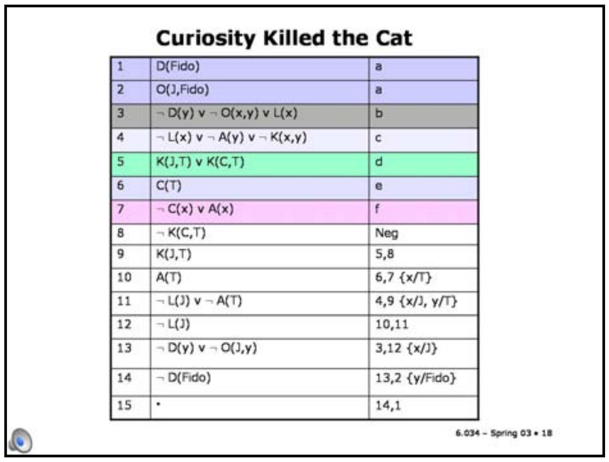
Slide 10.5.17
From 13 and 2, substituting Fido for x, we get **not D(Fido)**.



Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	¬ D(y) v ¬ O(x,y) v L(x)	b
4	¬ L(x) v ¬ A(y) v ¬ K(x,y)	c
5	K(J,T) v K(C,T)	d
6	C(T)	e
7	¬ C(x) v A(x)	f
8	¬ K(C,T)	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}
11	¬ L(J) v ¬ A(T)	4,9 {x/J, y/T}
12	¬ L(J)	10,11
13	¬ D(y) v ¬ O(J,y)	3,12 {x/J}
14	¬ D(Fido)	13,2 {y/Fido}

6.034 – Spring 03 • 17



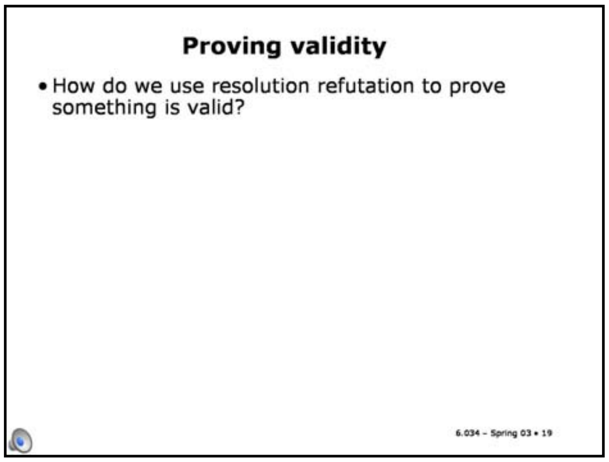
Curiosity Killed the Cat

1	D(Fido)	a
2	O(J,Fido)	a
3	¬ D(y) v ¬ O(x,y) v L(x)	b
4	¬ L(x) v ¬ A(y) v ¬ K(x,y)	c
5	K(J,T) v K(C,T)	d
6	C(T)	e
7	¬ C(x) v A(x)	f
8	¬ K(C,T)	Neg
9	K(J,T)	5,8
10	A(T)	6,7 {x/T}
11	¬ L(J) v ¬ A(T)	4,9 {x/J, y/T}
12	¬ L(J)	10,11
13	¬ D(y) v ¬ O(J,y)	3,12 {x/J}
14	¬ D(Fido)	13,2 {y/Fido}
15	•	14,1

6.034 – Spring 03 • 18

Slide 10.5.18
And finally, from lines 14 and 1, we derive a contradiction. Yay! Curiosity did kill the cat.

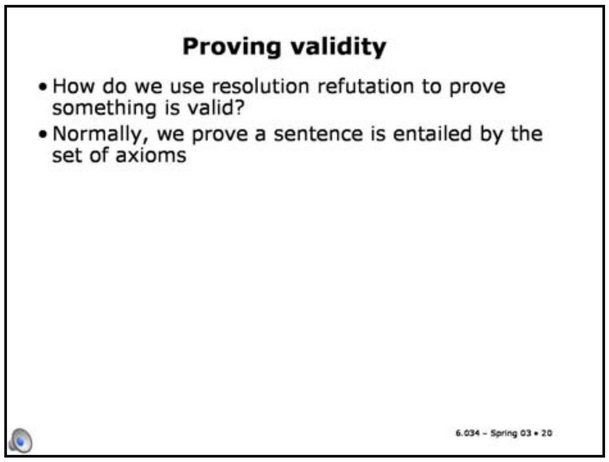
Slide 10.5.19
So, if we want to use resolution refutation to prove that something is valid, what would we do? What do we normally do when we do a proof using resolution refutation?



Proving validity

- How do we use resolution refutation to prove something is valid?

6.034 – Spring 03 • 19



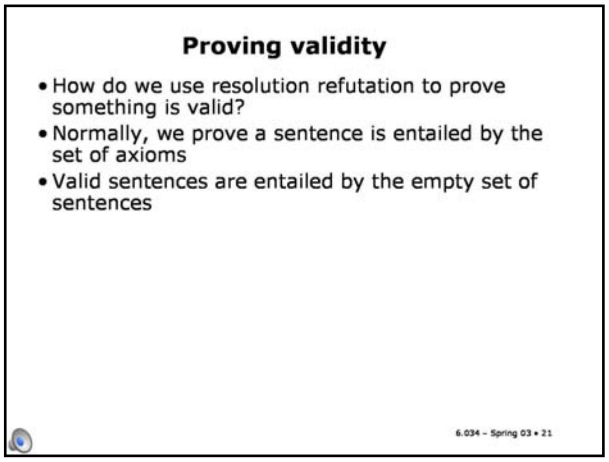
Proving validity

- How do we use resolution refutation to prove something is valid?
- Normally, we prove a sentence is entailed by the set of axioms

6.034 – Spring 03 • 20

Slide 10.5.20
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.

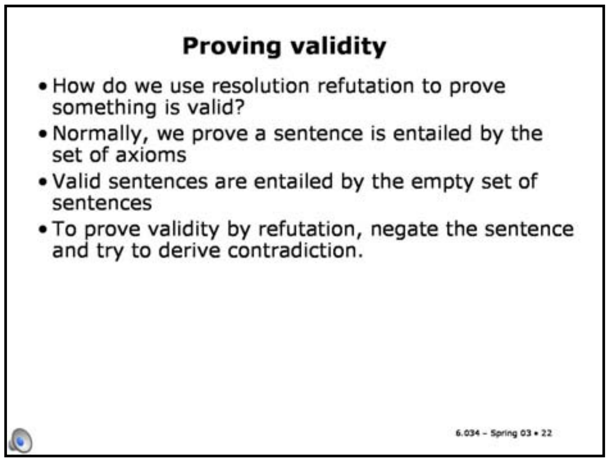
Slide 10.5.21
What does it mean for a sentence to be valid, in the language of entailment? That it's true in all interpretations. What that means really is that it should be derivable from nothing. A valid sentence is entailed by the empty set of sentences. The valid sentence is true no matter what. So we're going to prove something with no assumptions.



Proving validity

- How do we use resolution refutation to prove something is valid?
- Normally, we prove a sentence is entailed by the set of axioms
- Valid sentences are entailed by the empty set of sentences

6.034 – Spring 03 • 21



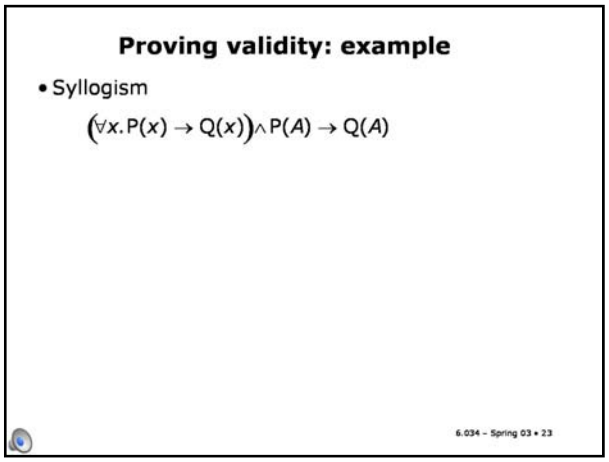
Proving validity

- How do we use resolution refutation to prove something is valid?
- Normally, we prove a sentence is entailed by the set of axioms
- Valid sentences are entailed by the empty set of sentences
- To prove validity by refutation, negate the sentence and try to derive contradiction.

6.034 – Spring 03 • 22

Slide 10.5.22
We can prove it by resolution refutation by negating the sentence and trying to derive a contradiction.

Slide 10.5.23
So, let's do an example. Imagine that we would like to show the validity of this sentence, which is a classical Aristotelian syllogism.

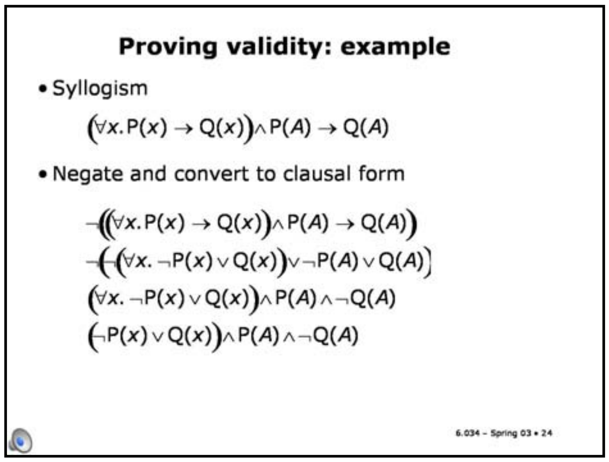


Proving validity: example

- Syllogism

$$\left(\forall x. P(x) \rightarrow Q(x)\right) \wedge P(A) \rightarrow Q(A)$$

6.034 – Spring 03 • 23



Proving validity: example

- Syllogism

$$(\forall x. P(x) \rightarrow Q(x)) \wedge P(A) \rightarrow Q(A)$$

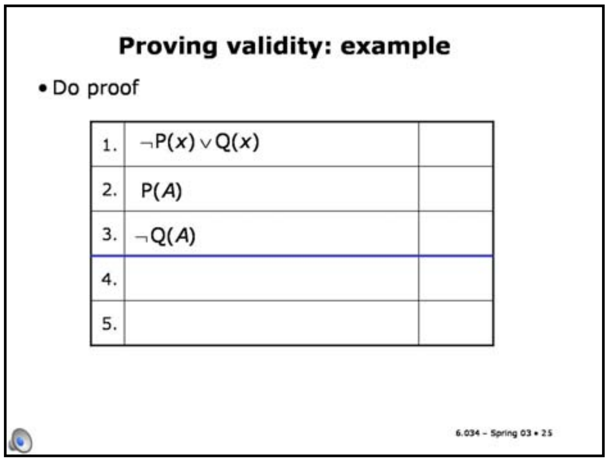
- Negate and convert to clausal form

$$\neg((\forall x. P(x) \rightarrow Q(x)) \wedge P(A) \rightarrow Q(A))$$
$$\neg(\neg(\forall x. \neg P(x) \vee Q(x)) \vee \neg P(A) \vee Q(A))$$
$$(\forall x. \neg P(x) \vee Q(x)) \wedge P(A) \wedge \neg Q(A)$$
$$(\neg P(x) \vee Q(x)) \wedge P(A) \wedge \neg Q(A)$$

6.034 – Spring 03 • 24

Slide 10.5.24
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.

Slide 10.5.25
We enter the clauses into our proof.

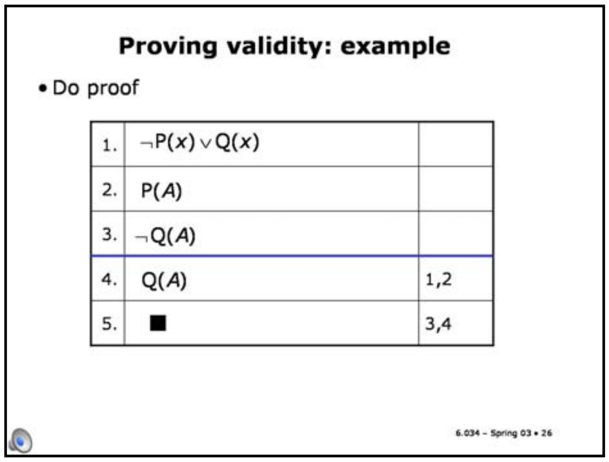


Proving validity: example

- Do proof

1.	$\neg P(x) \vee Q(x)$	
2.	$P(A)$	
3.	$\neg Q(A)$	
4.		
5.		

6.034 – Spring 03 • 25



Proving validity: example

- Do proof

1.	$\neg P(x) \vee Q(x)$	
2.	$P(A)$	
3.	$\neg Q(A)$	
4.	$Q(A)$	1,2
5.	■	3,4

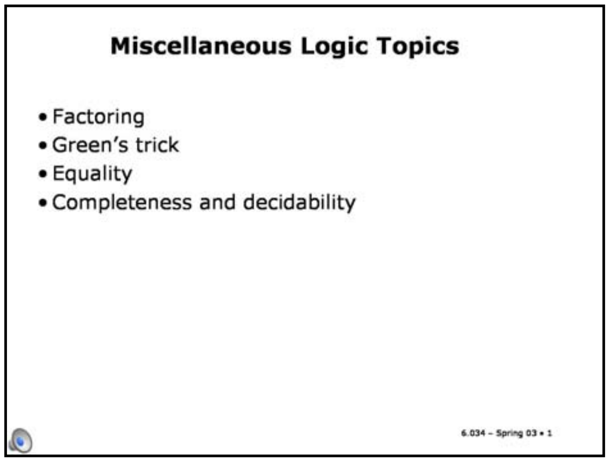
6.034 – Spring 03 • 26

Slide 10.5.26
Now, we can resolve lines 1 and 2, substituting A for X, and get Q(A).

And we can resolve 3 and 4, to get a contradiction.

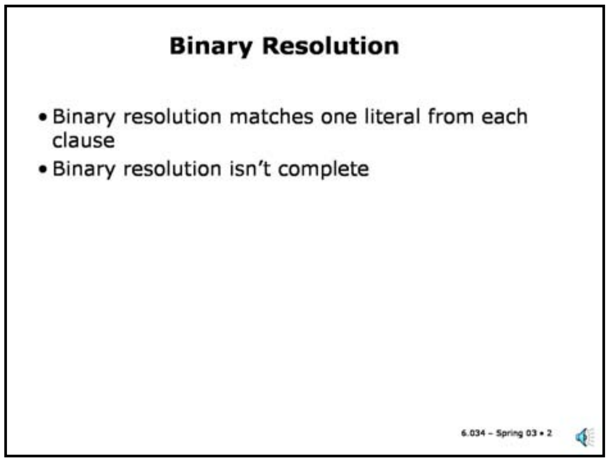
6.034 Notes: Section 10.6

Slide 10.6.1
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.


Miscellaneous Logic Topics

- Factoring
- Green's trick
- Equality
- Completeness and decidability

6.034 – Spring 03 • 1

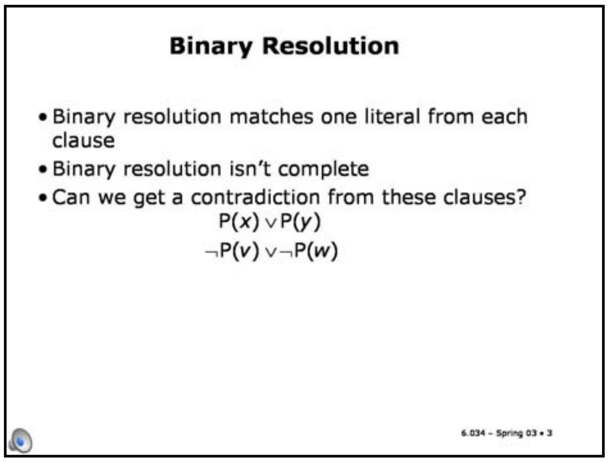

Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete

6.034 – Spring 03 • 2

Slide 10.6.2
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 for first-order logic. There are sets of unsatisfiable clauses that will not generate a contradiction by successive applications of binary resolution.

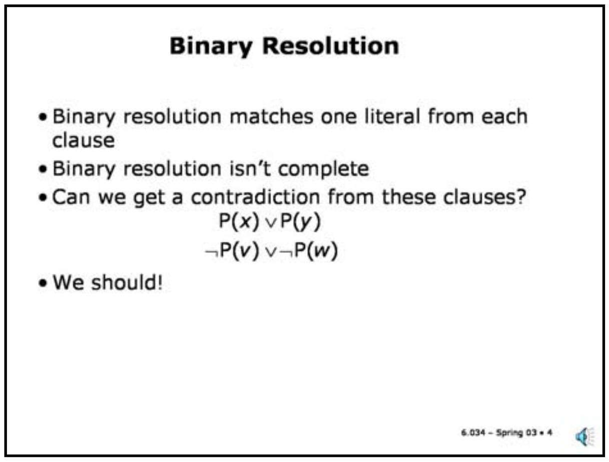
Slide 10.6.3
Here's a pair of clauses. **P(x) or P(y)** and **not P(v) or not P(w)**. Can we get a contradiction from them using binary resolution?


Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete
- Can we get a contradiction from these clauses?

$$P(x) \vee P(y)$$
$$\neg P(v) \vee \neg P(w)$$

6.034 – Spring 03 • 3


Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete
- Can we get a contradiction from these clauses?

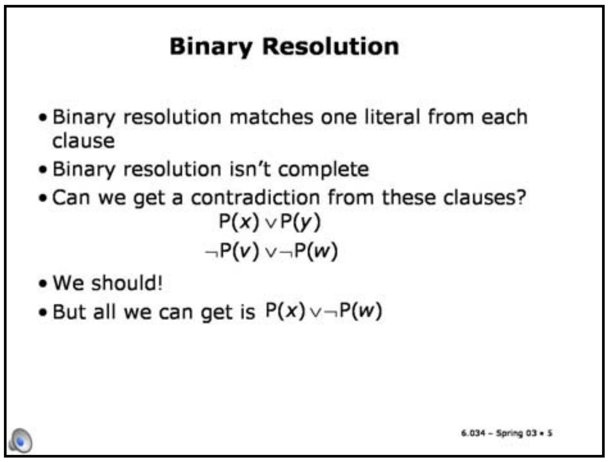
$$P(x) \vee P(y)$$
$$\neg P(v) \vee \neg P(w)$$

- We should!

6.034 – Spring 03 • 4

Slide 10.6.4
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(x) and not (P x)** to be true for everything in the universe.

Slide 10.6.5
But when we apply binary resolution to these clauses, all we can get is something like **P(x) or not P (w)** (your variables may vary).



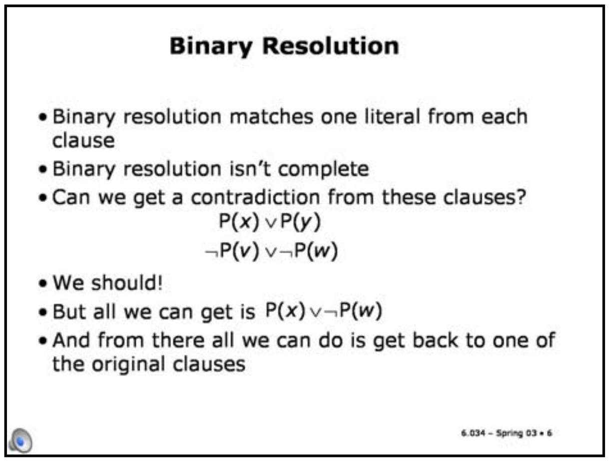
Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete
- Can we get a contradiction from these clauses?

$$P(x) \vee P(y)$$
$$\neg P(v) \vee \neg P(w)$$

- We should!
- But all we can get is $P(x) \vee \neg P(w)$

6.034 – Spring 03 • 5



Binary Resolution

- Binary resolution matches one literal from each clause
- Binary resolution isn't complete
- Can we get a contradiction from these clauses?

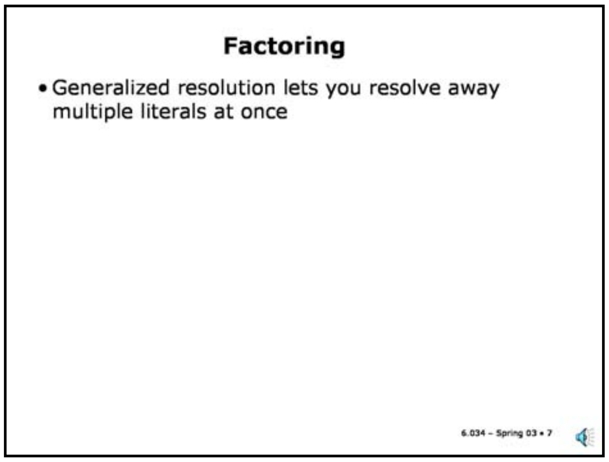
$$P(x) \vee P(y)$$
$$\neg P(v) \vee \neg P(w)$$

- We should!
- But all we can get is $P(x) \vee \neg P(w)$
- And from there all we can do is get back to one of the original clauses

6.034 – Spring 03 • 6

Slide 10.6.6
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.

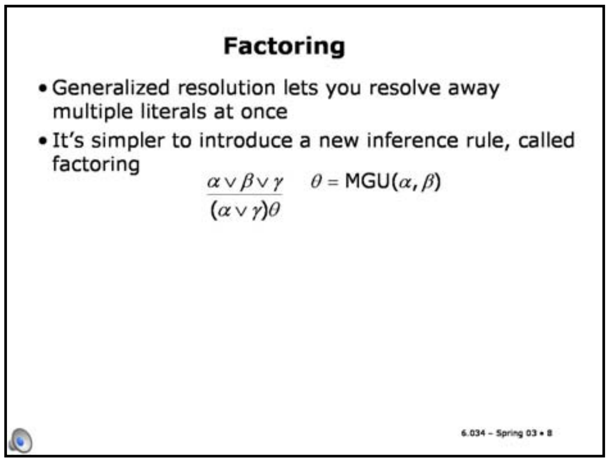
Slide 10.6.7
It turns out that there is a simple extension of binary resolution that is complete. In that version, known as generalized resolution, we look for subsets of literals in one clause that can be unified with the negation of a subset of literals in the other clause. In our example from before, each P literal in one clause can be unified with its negation in the other clause.



Factoring

- Generalized resolution lets you resolve away multiple literals at once

6.034 – Spring 03 • 7



Factoring

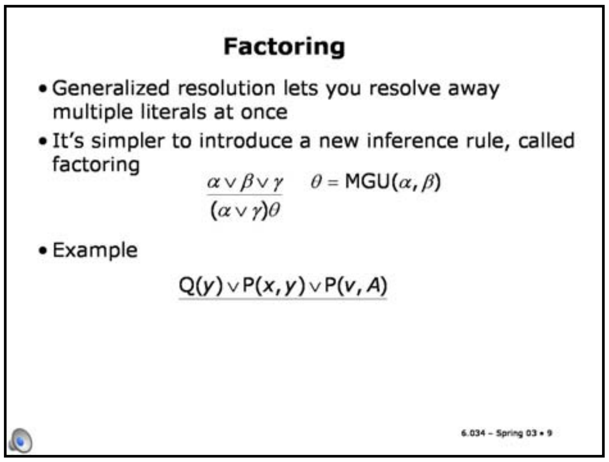
- Generalized resolution lets you resolve away multiple literals at once
- It's simpler to introduce a new inference rule, called factoring

$$\frac{\alpha \vee \beta \vee \gamma}{(\alpha \vee \gamma)\theta} \quad \theta = \mathrm{MGU}(\alpha, \beta)$$

6.034 – Spring 03 • 8

Slide 10.6.8
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.

Slide 10.6.9
So, for example, we can apply factoring to this sentence, by unifying P(x,y) and P(v,A).



Factoring

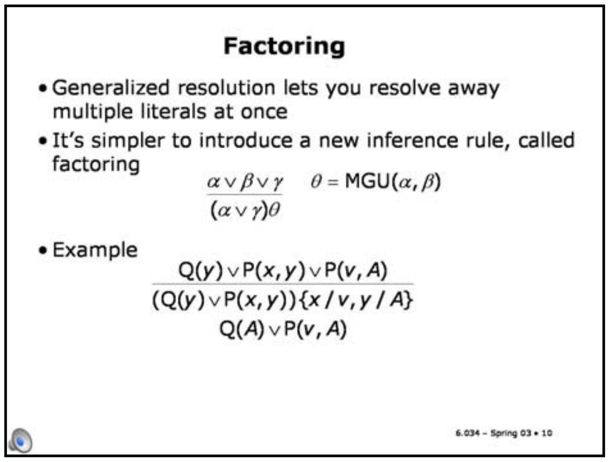
- Generalized resolution lets you resolve away multiple literals at once
- It's simpler to introduce a new inference rule, called factoring

$$\frac{\alpha \vee \beta \vee \gamma}{(\alpha \vee \gamma)\theta} \quad \theta = \text{MGU}(\alpha, \beta)$$

- Example

$$\frac{Q(y) \vee P(x,y) \vee P(v,A)}{}$$

6.034 – Spring 03 • 9



Factoring

- Generalized resolution lets you resolve away multiple literals at once
- It's simpler to introduce a new inference rule, called factoring

$$\frac{\alpha \vee \beta \vee \gamma}{(\alpha \vee \gamma)\theta} \quad \theta = \text{MGU}(\alpha, \beta)$$

- Example

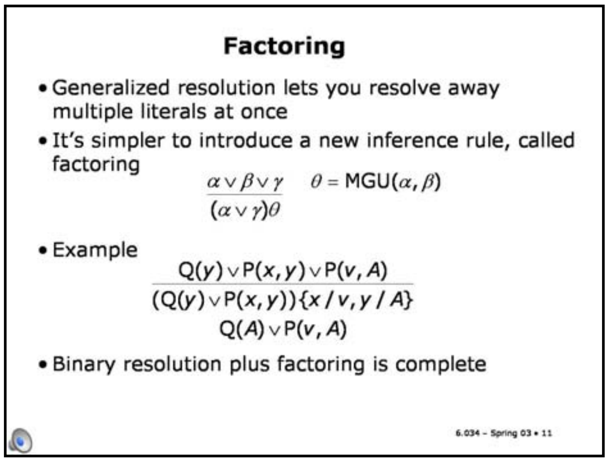
$$\frac{Q(y) \vee P(x,y) \vee P(v,A)}{(Q(y) \vee P(x,y))\{x/v, y/A\}}$$
$$Q(A) \vee P(v,A)$$

6.034 – Spring 03 • 10

Slide 10.6.10
We get **Q(Y) or P(x,Y)**, and then we have to apply the substitution {x/v, y/A}, which yields the result **Q(A) or 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.

Slide 10.6.11
And binary resolution, combined with factoring, is complete in a sense that we'll study more carefully later in this section.



Factoring

- Generalized resolution lets you resolve away multiple literals at once
- It's simpler to introduce a new inference rule, called factoring

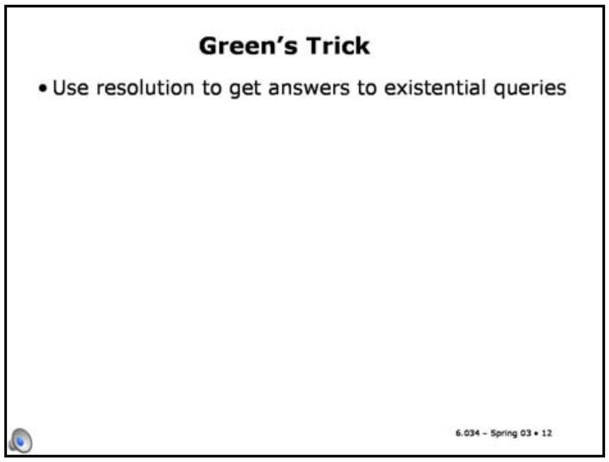
$$\frac{\alpha \vee \beta \vee \gamma}{(\alpha \vee \gamma)\theta} \quad \theta = \text{MGU}(\alpha, \beta)$$

- Example

$$\frac{Q(y) \vee P(x,y) \vee P(v,A)}{(Q(y) \vee P(x,y))\{x/v, y/A\}}$$
$$Q(A) \vee P(v,A)$$

- Binary resolution plus factoring is complete

6.034 – Spring 03 • 11



Green's Trick

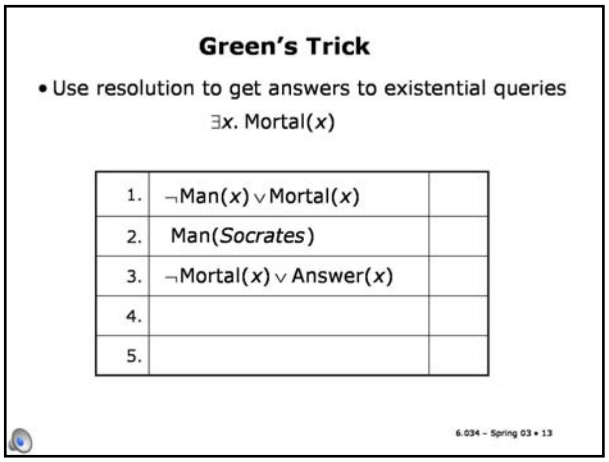
- Use resolution to get answers to existential queries

6.034 – Spring 03 • 12

Slide 10.6.12
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 actions 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 10.6.13
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)**.

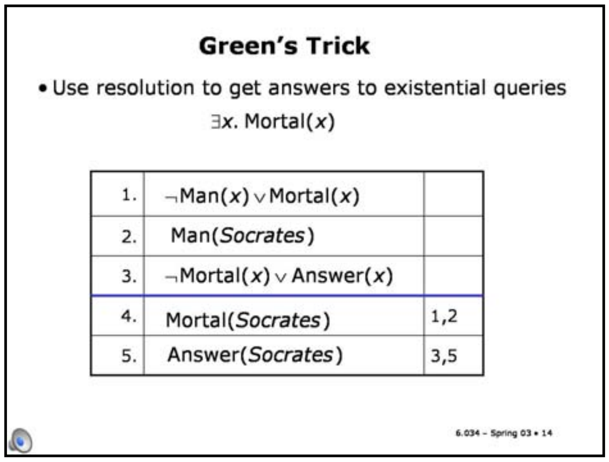


Green's Trick

- Use resolution to get answers to existential queries
  $\exists x.\ \text{Mortal}(x)$

1.	$\neg\text{Man}(x) \vee \text{Mortal}(x)$	
2.	$\text{Man}(\mathit{Socrates})$	
3.	$\neg\text{Mortal}(x) \vee \text{Answer}(x)$	
4.		
5.		

6.034 – Spring 03 • 13



Green's Trick

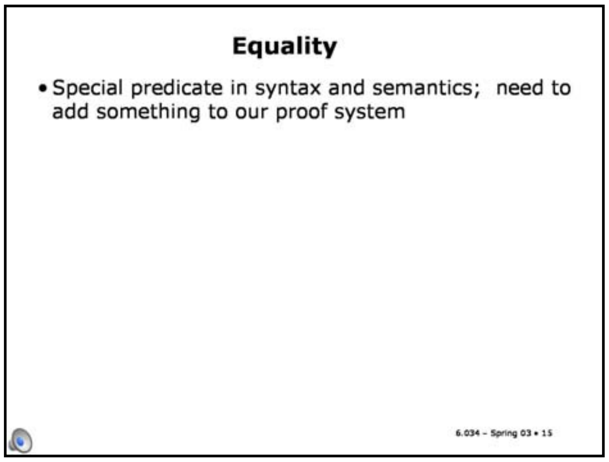
- Use resolution to get answers to existential queries
  $\exists x.\ \text{Mortal}(x)$

1.	$\neg\text{Man}(x) \vee \text{Mortal}(x)$	
2.	$\text{Man}(\mathit{Socrates})$	
3.	$\neg\text{Mortal}(x) \vee \text{Answer}(x)$	
4.	$\text{Mortal}(\mathit{Socrates})$	1,2
5.	$\text{Answer}(\mathit{Socrates})$	3,5

6.034 – Spring 03 • 14

Slide 10.6.14
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.

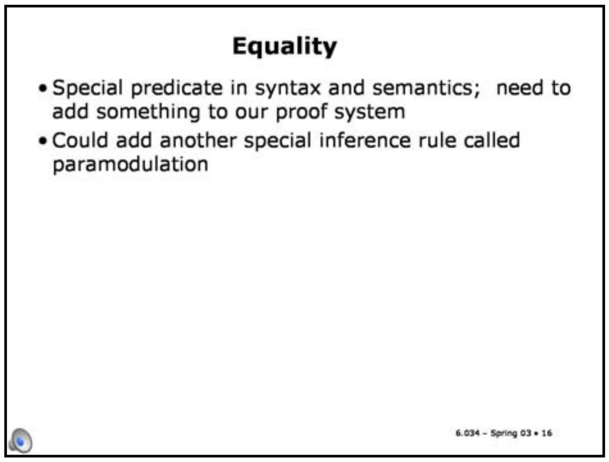
Slide 10.6.15
When we defined the language of first-order logic, we defined a special equality predicate. And we also defined special semantics for it (the sentence term1 equals term2 holds in an interpretation if and only if term1 and term2 both denote the same object in that interpretation). In order to do proofs that contain equality statements in them, we have to add a bit more mechanism.



Equality

- Special predicate in syntax and semantics; need to add something to our proof system

6.034 – Spring 03 • 15



Equality

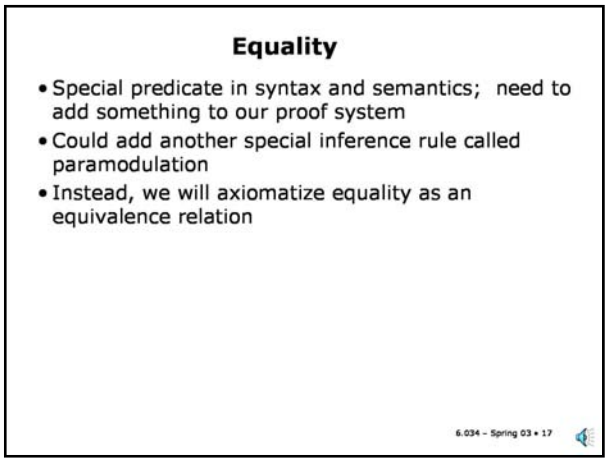
- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation

6.034 – Spring 03 • 16

Slide 10.6.16
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).

Slide 10.6.17
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.

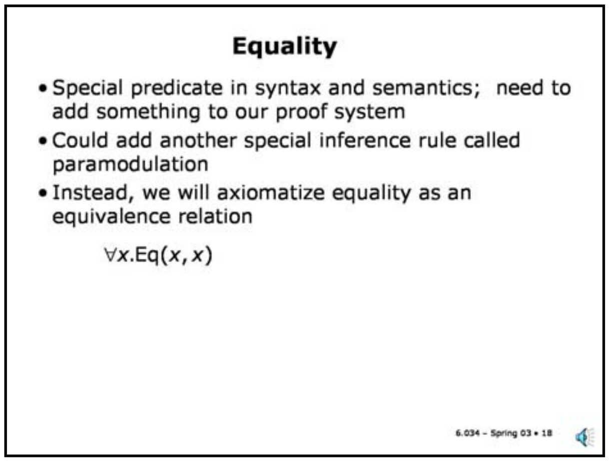
Just to make it clear that Equals, which we will write as Eq, is a predicate that we're going to handle normally in resolution, we'll write it with a word rather than the equals sign.



Equality

- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

6.034 – Spring 03 • 17



Equality

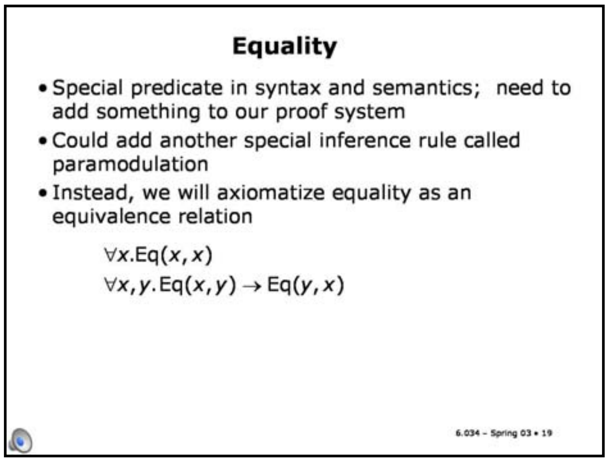
- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

$\forall x.\text{Eq}(x,x)$

6.034 – Spring 03 • 18

Slide 10.6.18
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.

Slide 10.6.19
Second, it's reflexive. If x is equal to y then y is equal to x.

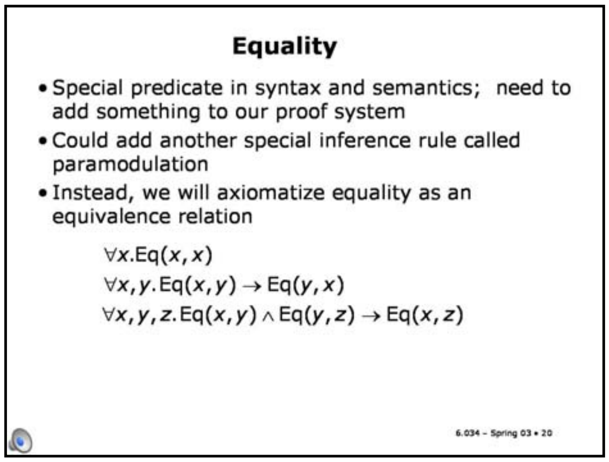


Equality

- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

$\forall x.\text{Eq}(x,x)$
$\forall x,y.\text{Eq}(x,y) \rightarrow \text{Eq}(y,x)$

6.034 – Spring 03 • 19



Equality

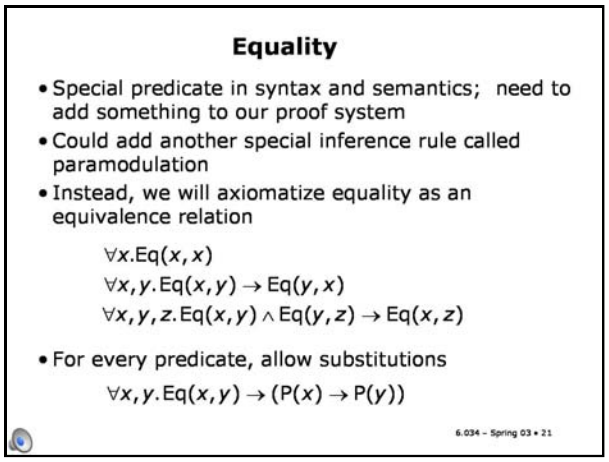
- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

$\forall x.\text{Eq}(x,x)$
$\forall x,y.\text{Eq}(x,y) \rightarrow \text{Eq}(y,x)$
$\forall x,y,z.\text{Eq}(x,y) \wedge \text{Eq}(y,z) \rightarrow \text{Eq}(x,z)$

6.034 – Spring 03 • 20

Slide 10.6.20
Third, it's transitive. That means that if x equals y and y equals z, then x equals z.

Slide 10.6.21
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.



Equality

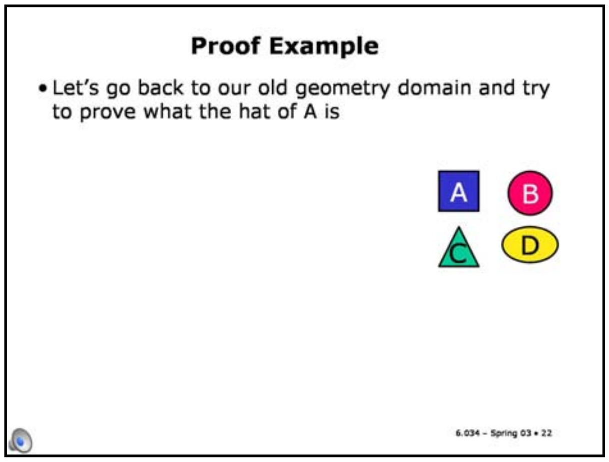
- Special predicate in syntax and semantics; need to add something to our proof system
- Could add another special inference rule called paramodulation
- Instead, we will axiomatize equality as an equivalence relation

$$\forall x.\mathrm{Eq}(x,x)$$
$$\forall x,y.\mathrm{Eq}(x,y) \rightarrow \mathrm{Eq}(y,x)$$
$$\forall x,y,z.\mathrm{Eq}(x,y) \wedge \mathrm{Eq}(y,z) \rightarrow \mathrm{Eq}(x,z)$$

- For every predicate, allow substitutions

$$\forall x,y.\mathrm{Eq}(x,y) \rightarrow (\mathrm{P}(x) \rightarrow \mathrm{P}(y))$$

6.034 – Spring 03 • 21



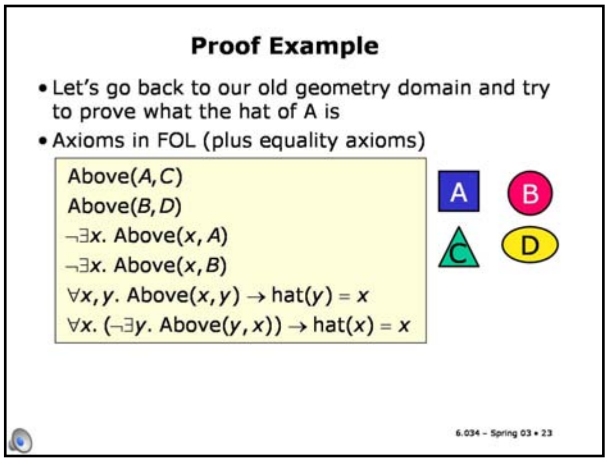
Proof Example

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

6.034 – Spring 03 • 22

Slide 10.6.22
Let's go back to our old geometry domain and try to prove what the hat of A is.

Slide 10.6.23
We know that these axioms (our old KB4) entail that hat(A) = A. We'll have to add in the equality axioms, as well.

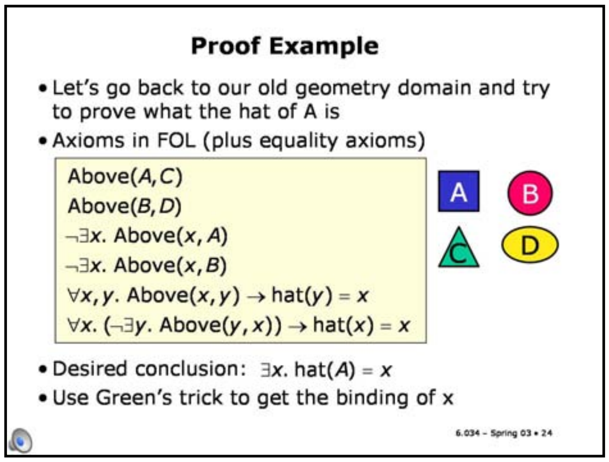


Proof Example

- 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)

$$\mathrm{Above}(A,C)$$
$$\mathrm{Above}(B,D)$$
$$\neg\exists x.\ \mathrm{Above}(x,A)$$
$$\neg\exists x.\ \mathrm{Above}(x,B)$$
$$\forall x,y.\ \mathrm{Above}(x,y) \rightarrow \mathrm{hat}(y) = x$$
$$\forall x.\ (\neg\exists y.\ \mathrm{Above}(y,x)) \rightarrow \mathrm{hat}(x) = x$$

6.034 – Spring 03 • 23



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)

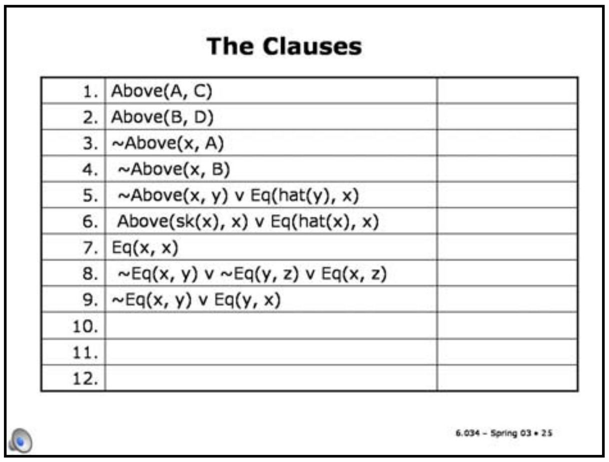
$$\mathrm{Above}(A,C)$$
$$\mathrm{Above}(B,D)$$
$$\neg\exists x.\ \mathrm{Above}(x,A)$$
$$\neg\exists x.\ \mathrm{Above}(x,B)$$
$$\forall x,y.\ \mathrm{Above}(x,y) \rightarrow \mathrm{hat}(y) = x$$
$$\forall x.\ (\neg\exists y.\ \mathrm{Above}(y,x)) \rightarrow \mathrm{hat}(x) = x$$

- Desired conclusion: $\exists x.\ \mathrm{hat}(A) = x$
- Use Green's trick to get the binding of x

6.034 – Spring 03 • 24

Slide 10.6.24
Let's see if we can derive that, using resolution refutation and Green's trick.

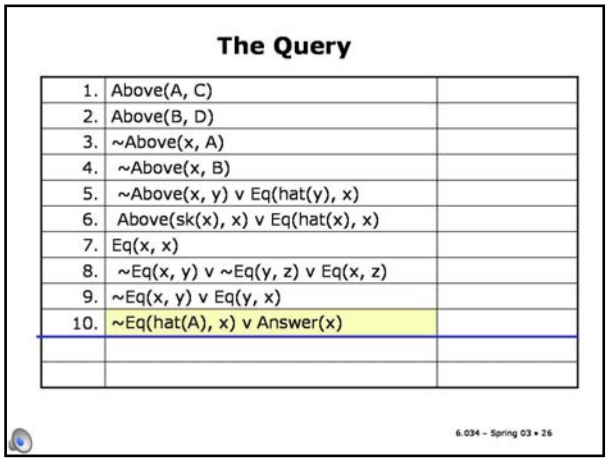
Slide 10.6.25
Here's the result of my clausal-form converter run on those axioms.



The Clauses

1.	Above(A, C)	
2.	Above(B, D)	
3.	~Above(x, A)	
4.	~Above(x, B)	
5.	~Above(x, y) v Eq(hat(y), x)	
6.	Above(sk(x), x) v Eq(hat(x), x)	
7.	Eq(x, x)	
8.	~Eq(x, y) v ~Eq(y, z) v Eq(x, z)	
9.	~Eq(x, y) v Eq(y, x)	
10.		
11.		
12.		

6.034 – Spring 03 • 25



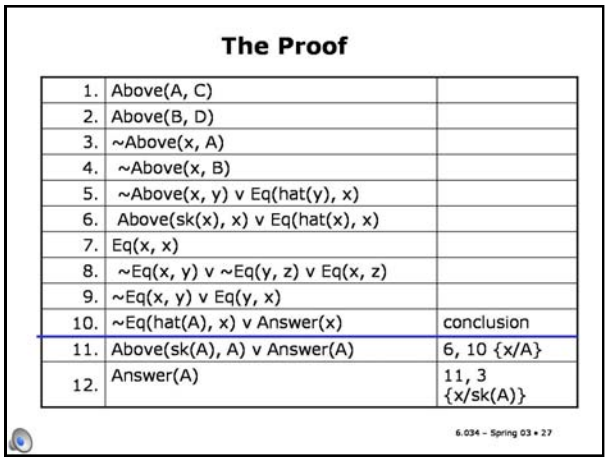
The Query

1.	Above(A, C)	
2.	Above(B, D)	
3.	~Above(x, A)	
4.	~Above(x, B)	
5.	~Above(x, y) v Eq(hat(y), x)	
6.	Above(sk(x), x) v Eq(hat(x), x)	
7.	Eq(x, x)	
8.	~Eq(x, y) v ~Eq(y, z) v Eq(x, z)	
9.	~Eq(x, y) v Eq(y, x)	
10.	~Eq(hat(A), x) v Answer(x)	

6.034 – Spring 03 • 26

Slide 10.6.26
Now, our goal is to prove **exists x such that Eq(hat(A),x)**. That is negated and turned into clausal form, yielding **not Eq(hat(A),x)**. And we add in the answer literal, so we can keep track of what the answer is.

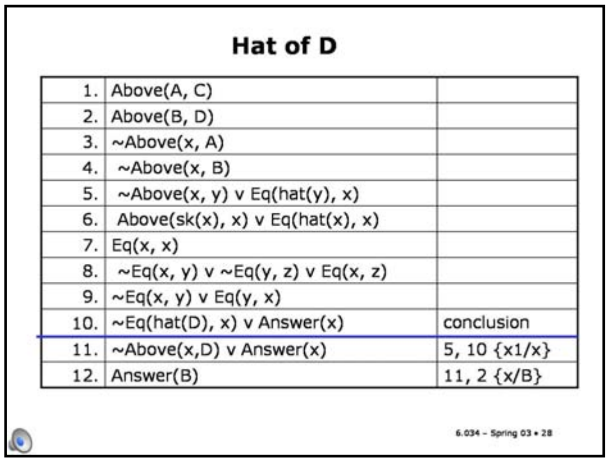
Slide 10.6.27
Here's the proof. The answer is A! And we figured this out without any kind of enumeration of interpretations.



The Proof

1.	Above(A, C)	
2.	Above(B, D)	
3.	~Above(x, A)	
4.	~Above(x, B)	
5.	~Above(x, y) v Eq(hat(y), x)	
6.	Above(sk(x), x) v Eq(hat(x), x)	
7.	Eq(x, x)	
8.	~Eq(x, y) v ~Eq(y, z) v Eq(x, z)	
9.	~Eq(x, y) v Eq(y, x)	
10.	~Eq(hat(A), x) v Answer(x)	conclusion
11.	Above(sk(A), A) v Answer(A)	6, 10 {x/A}
12.	Answer(A)	11, 3 {x/sk(A)}

6.034 – Spring 03 • 27



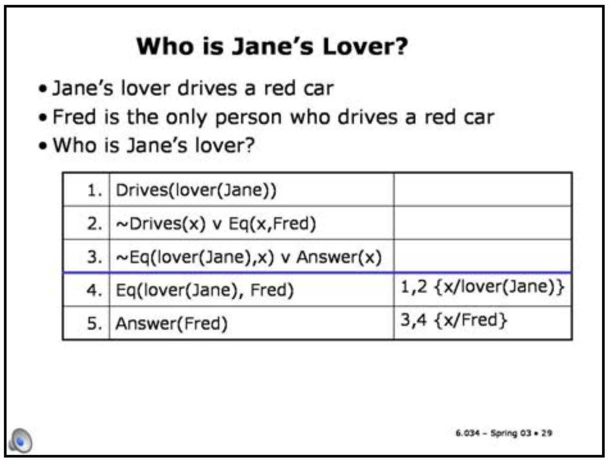
Hat of D

1.	Above(A, C)	
2.	Above(B, D)	
3.	~Above(x, A)	
4.	~Above(x, B)	
5.	~Above(x, y) v Eq(hat(y), x)	
6.	Above(sk(x), x) v Eq(hat(x), x)	
7.	Eq(x, x)	
8.	~Eq(x, y) v ~Eq(y, z) v Eq(x, z)	
9.	~Eq(x, y) v Eq(y, x)	
10.	~Eq(hat(D), x) v Answer(x)	conclusion
11.	~Above(x,D) v Answer(x)	5, 10 {x1/x}
12.	Answer(B)	11, 2 {x/B}

6.034 – Spring 03 • 28

Slide 10.6.28
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 10.6.29
Here'a a worked example of a problem with equality.

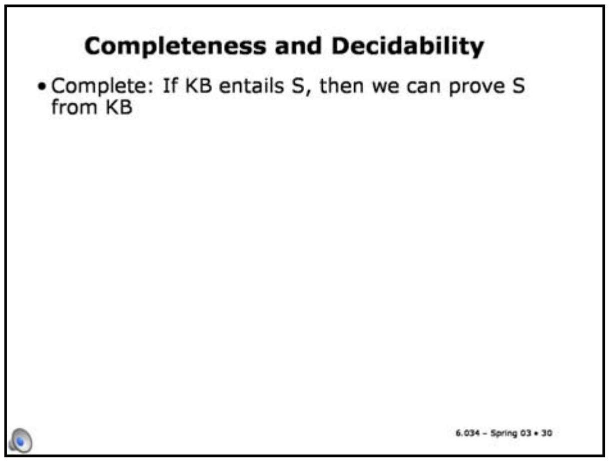


Who is Jane's Lover?

- Jane's lover drives a red car
- Fred is the only person who drives a red car
- Who is Jane's lover?

1.	Drives(lover(Jane))	
2.	~Drives(x) v Eq(x,Fred)	
3.	~Eq(lover(Jane),x) v Answer(x)	
4.	Eq(lover(Jane), Fred)	1,2 {x/lover(Jane)}
5.	Answer(Fred)	3,4 {x/Fred}

6.034 – Spring 03 • 29



Completeness and Decidability

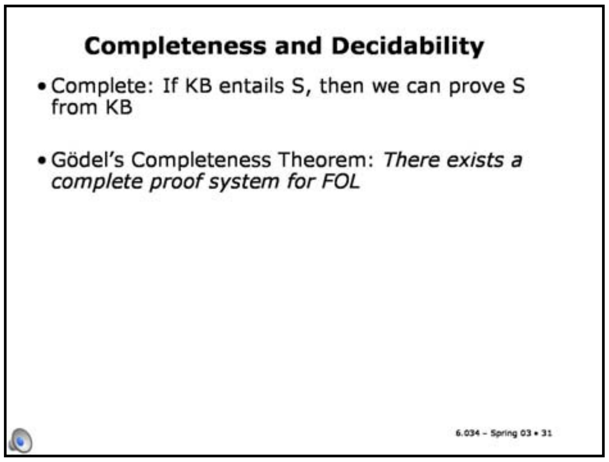
- Complete: If KB entails S, then we can prove S from KB

6.034 – Spring 03 • 30

Slide 10.6.30
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.

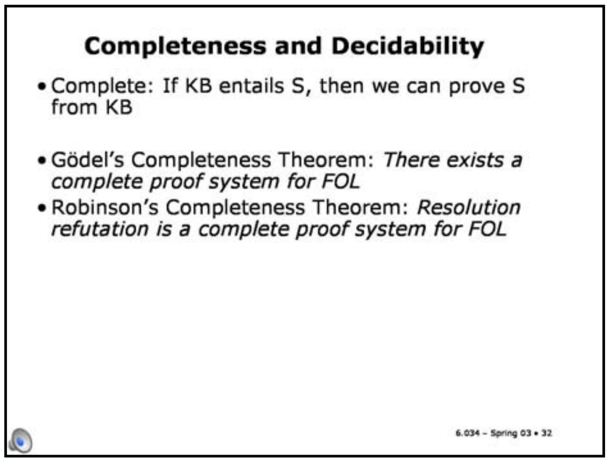
Slide 10.6.31
In 1929, Godel proved a completeness theorem for first-order logic: There exists a complete proof system for FOL. But, living up to his nature as a very abstract logician, he didn't come up with such a proof system; he just proved one existed.



Completeness and Decidability

- Complete: If KB entails S, then we can prove S from KB
- Gödel's Completeness Theorem: *There exists a complete proof system for FOL*

6.034 – Spring 03 • 31



Completeness and Decidability

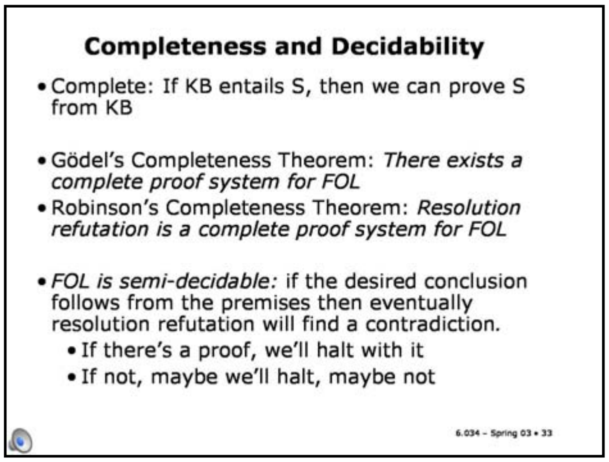
- Complete: If KB entails S, then we can prove S from KB
- Gödel's Completeness Theorem: *There exists a complete proof system for FOL*
- Robinson's Completeness Theorem: *Resolution refutation is a complete proof system for FOL*

6.034 – Spring 03 • 32

Slide 10.6.32
Then, in 1965, Robinson came along and showed that resolution refutation is sound and complete for FOL.

Slide 10.6.33
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.

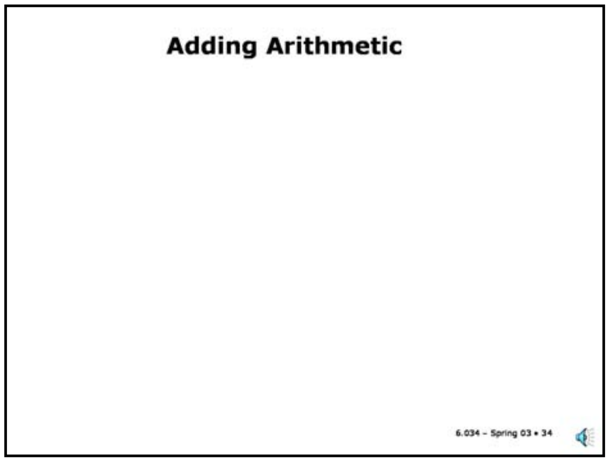
This makes first-order logic what is known as "semi-decidable". If the answer is "yes", that is, if there is a proof, then the theorem prover will eventually halt and say so. But if there isn't a proof, it might run forever!



Completeness and Decidability

- Complete: If KB entails S, then we can prove S from KB
- Gödel's Completeness Theorem: *There exists a complete proof system for FOL*
- Robinson's Completeness Theorem: *Resolution refutation is a complete proof system for FOL*
- *FOL is semi-decidable:* if the desired conclusion follows from the premises then eventually resolution refutation will find a contradiction.
  - If there's a proof, we'll halt with it
  - If not, maybe we'll halt, maybe not

6.034 – Spring 03 • 33

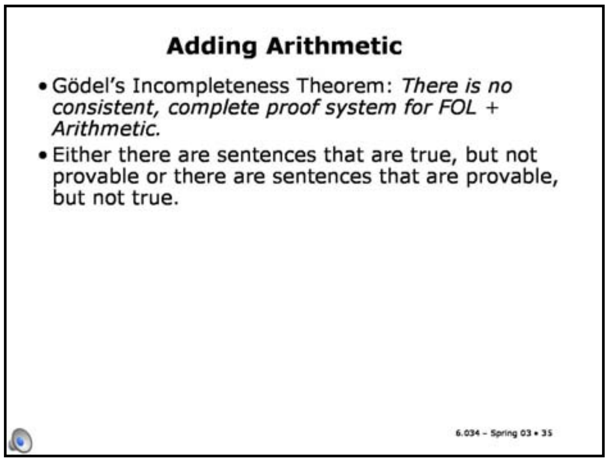


Adding Arithmetic

6.034 – Spring 03 • 34

Slide 10.6.34
So, things are relatively good with regular first-order logic. And they're still fine if you add addition to the language, allowing statements like **P(x) and (x + 2 = 3)**. But if you add addition and multiplication, it starts to get weird!

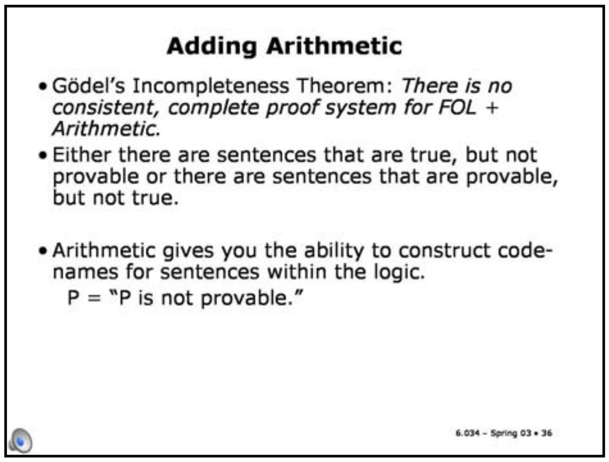
Slide 10.6.35
In 1931, Godel proved an incompleteness theorem, which says that there is no consistent, complete proof system for FOL plus arithmetic. (Consistent is the same as sound.) Either there are sentences that are true, but not provable, or there are sentences that are provable, but not true. It's not so good either way.



Adding Arithmetic

- Gödel's Incompleteness Theorem: *There is no consistent, complete proof system for FOL + Arithmetic.*
- Either there are sentences that are true, but not provable or there are sentences that are provable, but not true.

6.034 – Spring 03 • 35



Adding Arithmetic

- Gödel's Incompleteness Theorem: *There is no consistent, complete proof system for FOL + Arithmetic.*
- Either there are sentences that are true, but not provable or there are sentences that are provable, but not true.
- Arithmetic gives you the ability to construct code-names for sentences within the logic.
  P = "P is not provable."

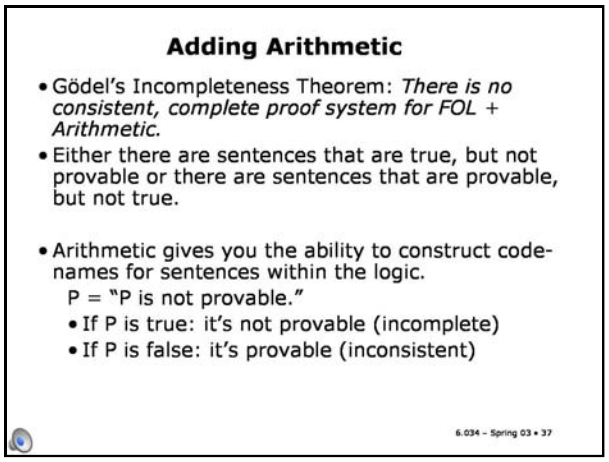
6.034 – Spring 03 • 36

Slide 10.6.36
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 10.6.37
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.



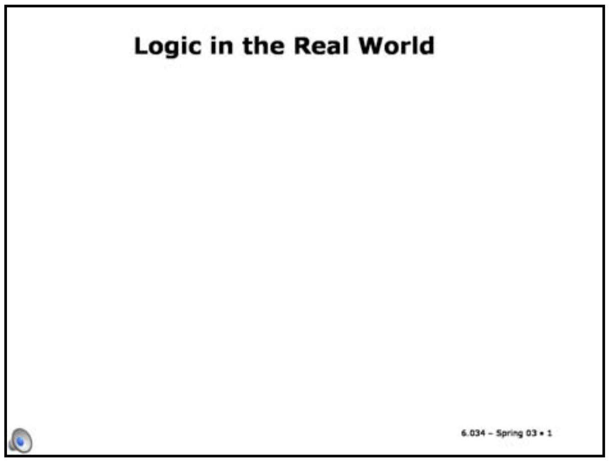
Adding Arithmetic

- Gödel's Incompleteness Theorem: *There is no consistent, complete proof system for FOL + Arithmetic.*
- Either there are sentences that are true, but not provable or there are sentences that are provable, but not true.
- Arithmetic gives you the ability to construct code-names for sentences within the logic.
  P = "P is not provable."
  - If P is true: it's not provable (incomplete)
  - If P is false: it's provable (inconsistent)

6.034 – Spring 03 • 37

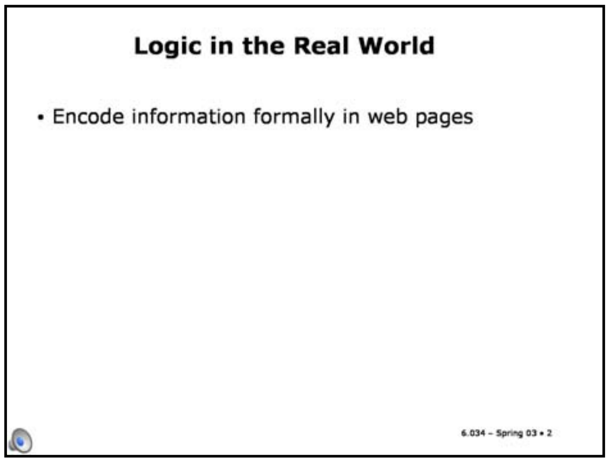
6.034 Notes: Section 10.7

Slide 10.7.1
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.



Logic in the Real World

6.034 – Spring 03 • 1



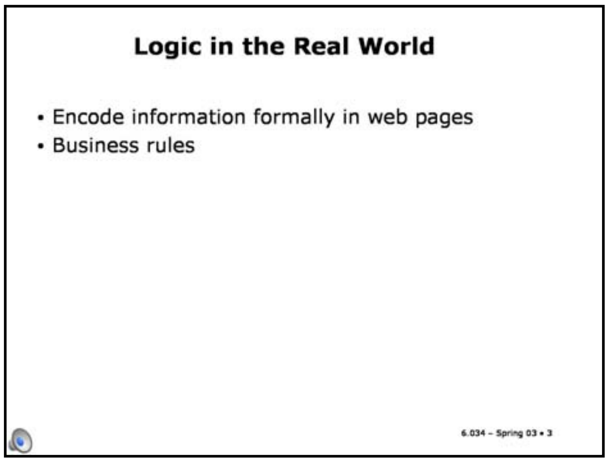
Logic in the Real World

- Encode information formally in web pages

6.034 – Spring 03 • 2

Slide 10.7.2
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.

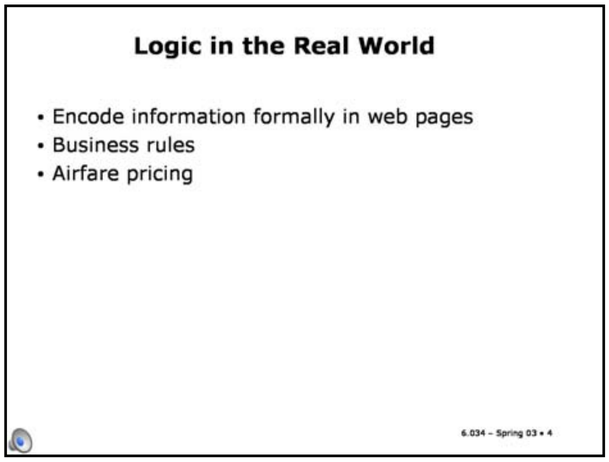
Slide 10.7.3
It is becoming more appealing, in business, to have computers talk directly to one another, and to leave humans out of the loop. One place this can happen is in negotiating simple contracts between companies to deliver goods at some price. Benjamin Grosof, who is a professor in the Sloan 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.



Logic in the Real World

- Encode information formally in web pages
- Business rules

6.034 – Spring 03 • 3



Logic in the Real World

- Encode information formally in web pages
- Business rules
- Airfare pricing

6.034 – Spring 03 • 4

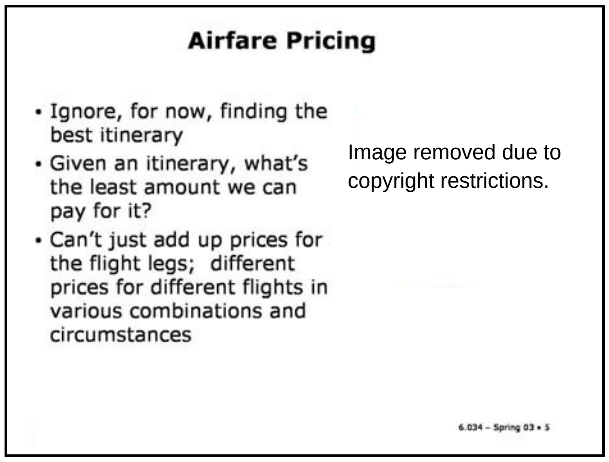
Slide 10.7.4
Another example, which we'll pursue in detail, is the language the airlines use to specify the rules on their airfares. It turns out that every day, many times a day, airlines revise and publish (electronically) their fare structures. And, as many of you know, the rules governing the pricing 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 10.7.5
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.

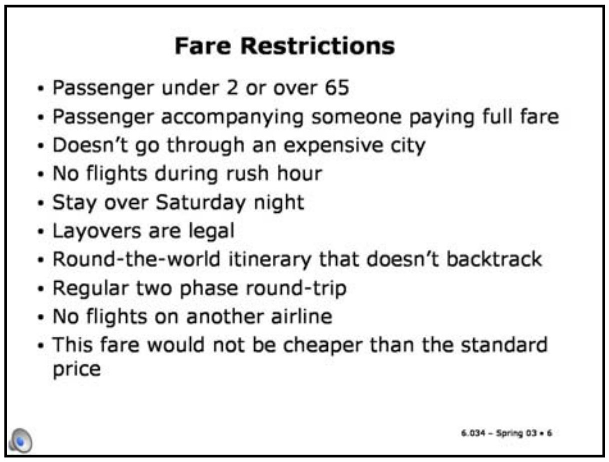


Airfare Pricing

- Ignore, for now, finding the best itinerary
- Given an itinerary, what's the least amount we can pay for it?
- Can't just add up prices for the flight legs; different prices for different flights in various combinations and circumstances

Image removed due to copyright restrictions.

6.034 – Spring 03 • 5



Fare Restrictions

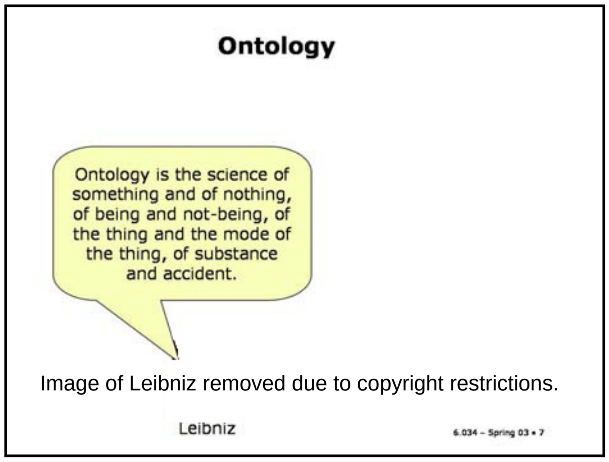
- Passenger under 2 or over 65
- Passenger accompanying someone paying full fare
- Doesn't go through an expensive city
- No flights during rush hour
- Stay over Saturday night
- Layovers are legal
- Round-the-world itinerary that doesn't backtrack
- Regular two phase round-trip
- No flights on another airline
- This fare would not be cheaper than the standard price

6.034 – Spring 03 • 6

Slide 10.7.6
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 10.7.7
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.

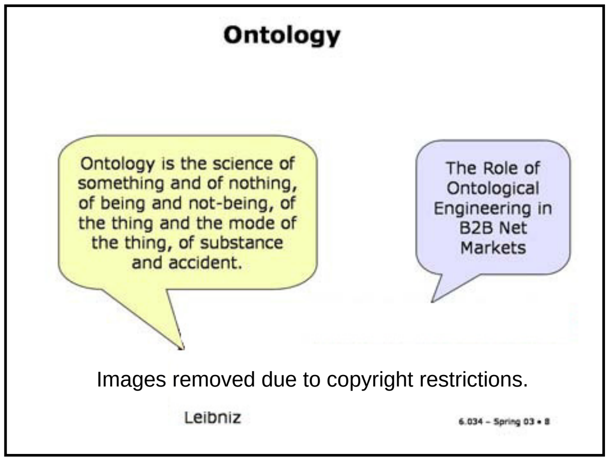

Ontology

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.

Image of Leibniz removed due to copyright restrictions.

Leibniz

6.034 – Spring 03 • 7


Ontology

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.

The Role of Ontological Engineering in B2B Net Markets

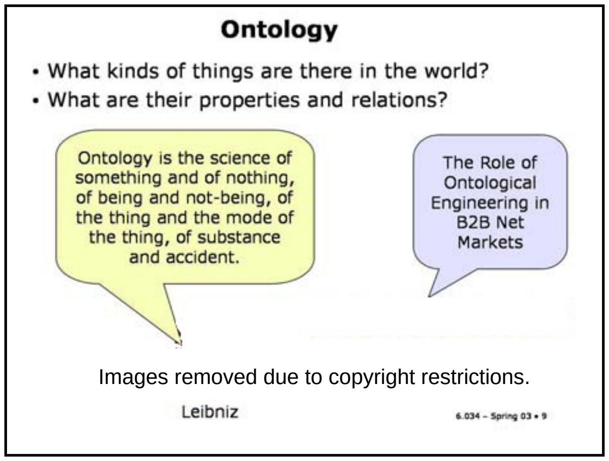
Images removed due to copyright restrictions.

Leibniz

6.034 – Spring 03 • 8

Slide 10.7.8
Now there are web sites with paper titles like "The Role of Ontological Engineering in B2B Net Markets". That's just as scary.

Slide 10.7.9
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.


Ontology

- What kinds of things are there in the world?
- What are their properties and relations?

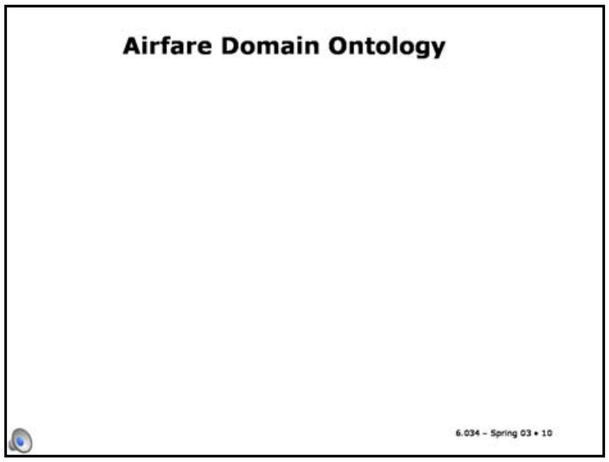
Ontology is the science of something and of nothing, of being and not-being, of the thing and the mode of the thing, of substance and accident.

The Role of Ontological Engineering in B2B Net Markets

Images removed due to copyright restrictions.

Leibniz

6.034 – Spring 03 • 9

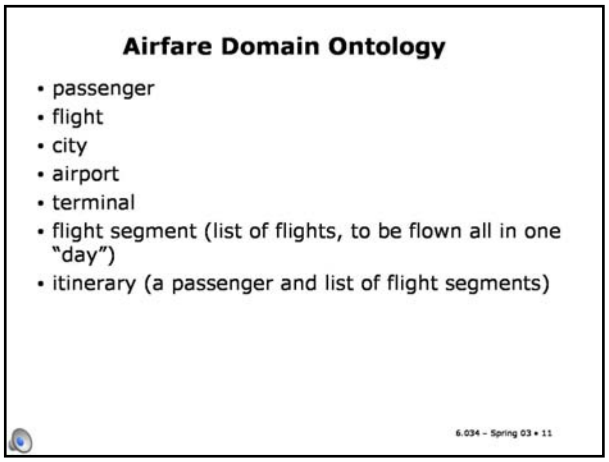

Airfare Domain Ontology

6.034 – Spring 03 • 10

Slide 10.7.10
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.

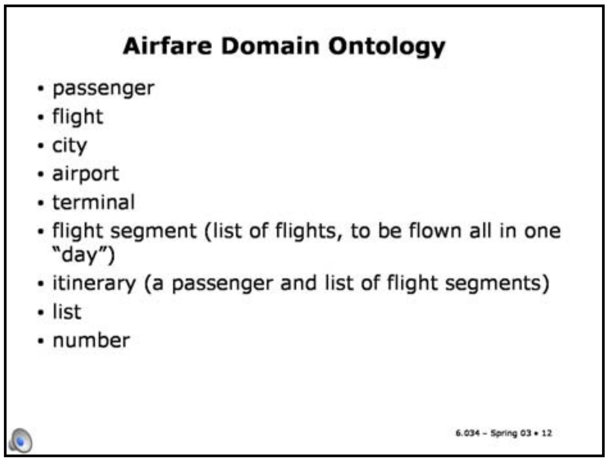
Slide 10.7.11
Here's a list of the relevant object types I came up with:

- 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)



Airfare Domain Ontology

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



Airfare Domain Ontology

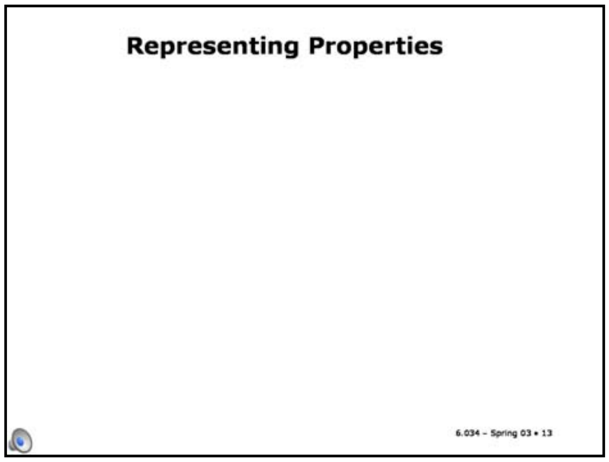
- passenger
- flight
- city
- airport
- terminal
- flight segment (list of flights, to be flown all in one "day")
- itinerary (a passenger and list of flight segments)
- list
- number

Slide 10.7.12
We'll also need some non-concrete object types, including

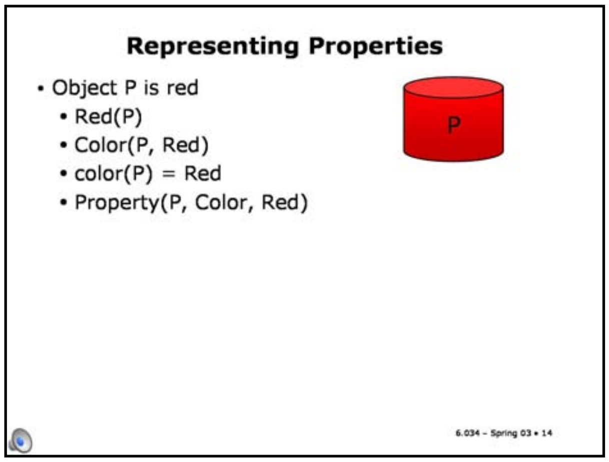
- list
- number

Slide 10.7.13
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.



Representing Properties



Representing Properties

- Object P is red
  - Red(P)
  - Color(P, Red)
  - color(P) = Red
  - Property(P, Color, Red)

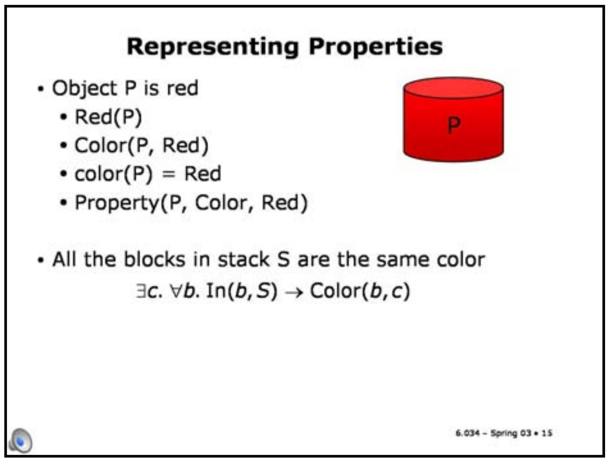
Slide 10.7.14
Let's talk, for a minute, about something simple, like saying that an object named P is red. There are a number of ways to say this, including:

- `Red(P)`
- `Color(P, Red)`
- `color(P) = Red`
- `Property(P, Color, Red)`

Slide 10.7.15
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. all b. In(b,S) -> Color(b, c)
```

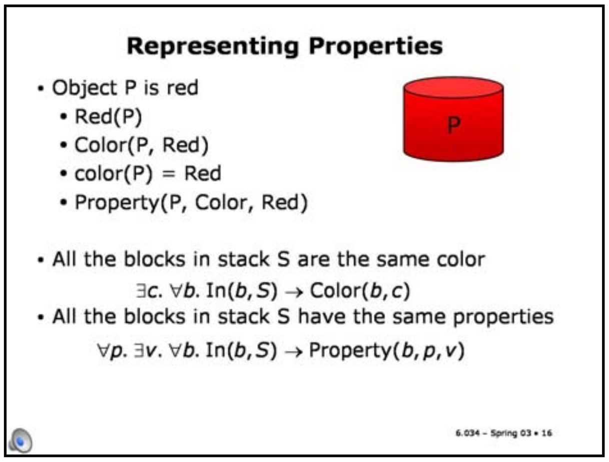
In this case, we have *reified* redness; that is, we've made it into an object that can be named and quantified over. It will turn out that it's often useful to use this kind of representation.



Representing Properties

- Object P is red
  - Red(P)
  - Color(P, Red)
  - color(P) = Red
  - Property(P, Color, Red)
- All the blocks in stack S are the same color
  $\exists c.\ \forall b.\ \text{In}(b,S) \rightarrow \text{Color}(b,c)$

6.034 – Spring 03 • 15



Representing Properties

- Object P is red
  - Red(P)
  - Color(P, Red)
  - color(P) = Red
  - Property(P, Color, Red)
- All the blocks in stack S are the same color
  $\exists c.\ \forall b.\ \text{In}(b,S) \rightarrow \text{Color}(b,c)$
- All the blocks in stack S have the same properties
  $\forall p.\ \exists v.\ \forall b.\ \text{In}(b,S) \rightarrow \text{Property}(b,p,v)$

6.034 – Spring 03 • 16

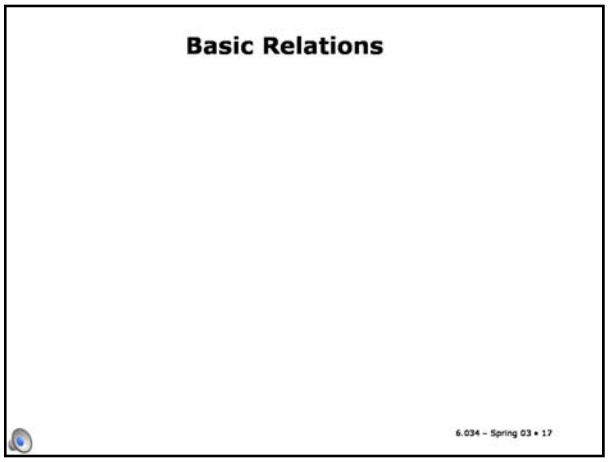
Slide 10.7.16
It's possible to go even farther down this road, in case, for instance, we wanted to say that all the blocks in stack S have all the same properties:

```
all p. exists v. all b. In(b,S) -> 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.

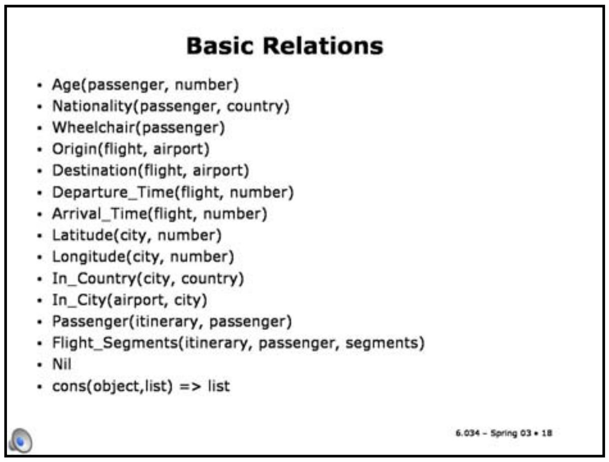
Some people advocate writing all logical specifications using this kind of formalization, because it is very general; but it's also kind of hard to read. We'll stick to representations closer to `Color(P, Red)`.

Slide 10.7.17
The particular relations we'll use come from two sources. Some relations will be used to specify the basic facts in our knowledge-base.



Basic Relations

6.034 – Spring 03 • 17



Basic Relations

- Age(passenger, number)
- Nationality(passenger, country)
- Wheelchair(passenger)
- Origin(flight, airport)
- Destination(flight, airport)
- Departure_Time(flight, number)
- Arrival_Time(flight, number)
- Latitude(city, number)
- Longitude(city, number)
- In_Country(city, country)
- In_City(airport, city)
- Passenger(itinerary, passenger)
- Flight_Segments(itinerary, passenger, segments)
- Nil
- cons(object,list) => list

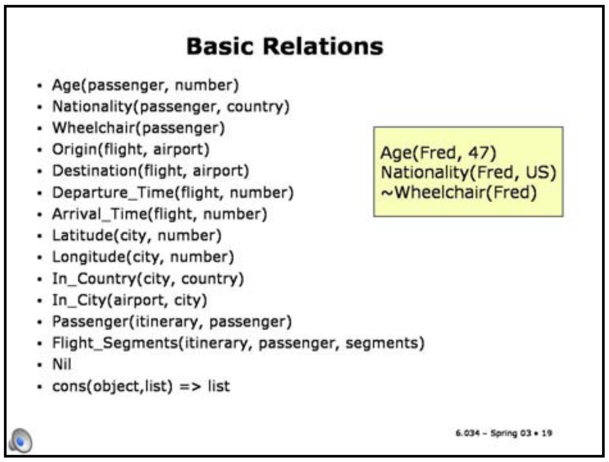
6.034 – Spring 03 • 18

Slide 10.7.18
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).

Slide 10.7.19
So, we might describe passenger Fred using the sentences

```
Age(Fred, 47)
Nationality(Fred, US)
~Wheelchair(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 so on.

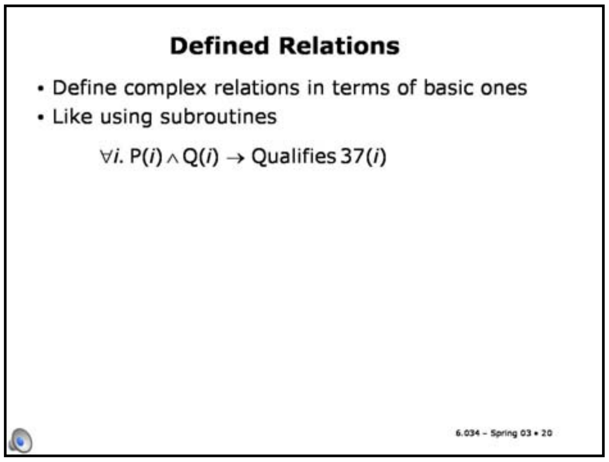


Basic Relations

- Age(passenger, number)
- Nationality(passenger, country)
- Wheelchair(passenger)
- Origin(flight, airport)
- Destination(flight, airport)
- Departure_Time(flight, number)
- Arrival_Time(flight, number)
- Latitude(city, number)
- Longitude(city, number)
- In_Country(city, country)
- In_City(airport, city)
- Passenger(itinerary, passenger)
- Flight_Segments(itinerary, passenger, segments)
- Nil
- cons(object,list) => list

Age(Fred, 47)
Nationality(Fred, US)
~Wheelchair(Fred)

6.034 – Spring 03 • 19



Defined Relations

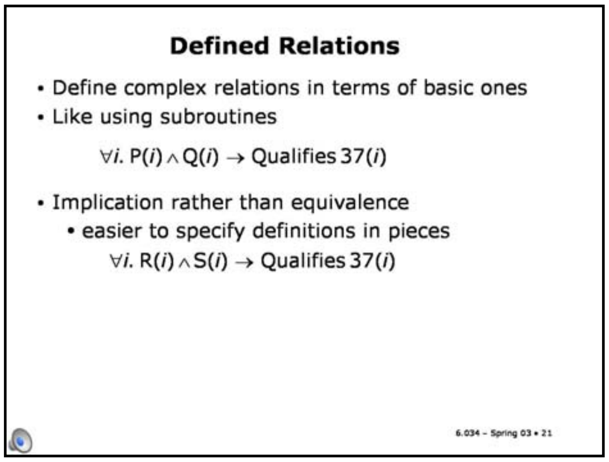
- Define complex relations in terms of basic ones
- Like using subroutines

$$\forall i.\ P(i) \wedge Q(i) \rightarrow \text{Qualifies}37(i)$$

6.034 – Spring 03 • 20

Slide 10.7.20
Other relations will be used to express the things we want to infer. An example in this domain might be `Qualifies_for_fare_class_37` or something else equally intuitive. As we begin trying to write down a logical expression for `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.

Slide 10.7.21
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).



Defined Relations

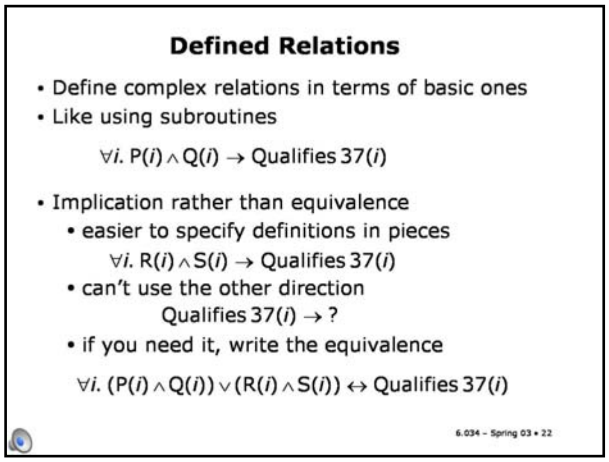
- Define complex relations in terms of basic ones
- Like using subroutines

$$\forall i.\ P(i) \wedge Q(i) \rightarrow \text{Qualifies}37(i)$$

- Implication rather than equivalence
  - easier to specify definitions in pieces

$$\forall i.\ R(i) \wedge S(i) \rightarrow \text{Qualifies}37(i)$$

6.034 – Spring 03 • 21



Defined Relations

- Define complex relations in terms of basic ones
- Like using subroutines

$$\forall i.\ P(i) \wedge Q(i) \rightarrow \text{Qualifies}37(i)$$

- Implication rather than equivalence
  - easier to specify definitions in pieces

$$\forall i.\ R(i) \wedge S(i) \rightarrow \text{Qualifies}37(i)$$

  - can't use the other direction

$$\text{Qualifies}37(i) \rightarrow ?$$

  - if you need it, write the equivalence

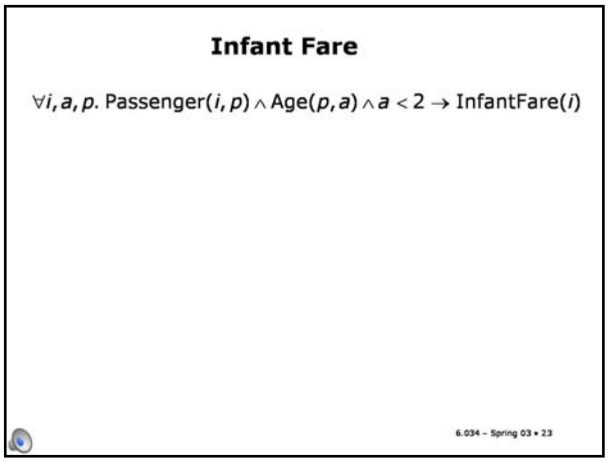
$$\forall i.\ (P(i) \wedge Q(i)) \vee (R(i) \wedge S(i)) \leftrightarrow \text{Qualifies}37(i)$$

6.034 – Spring 03 • 22

Slide 10.7.22
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.

Slide 10.7.23
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

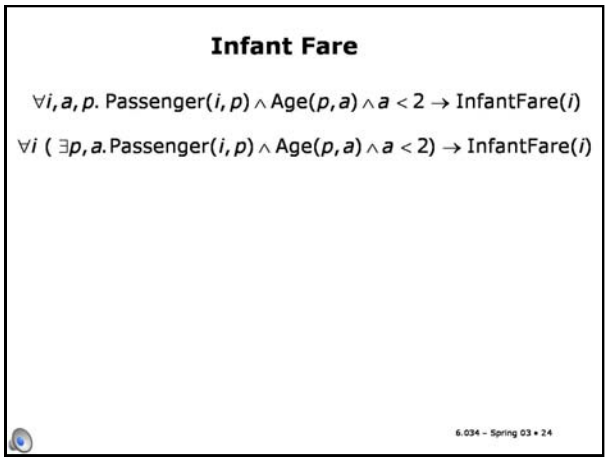
```
all i, a, p. Passenger(i,p) ^ Age(p,a) ^ a < 2  -> Infant_Fare(i)
```



Infant Fare

$\forall i, a, p.\ \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{InfantFare}(i)$

6.034 – Spring 03 • 23



Infant Fare

$\forall i, a, p.\ \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{InfantFare}(i)$

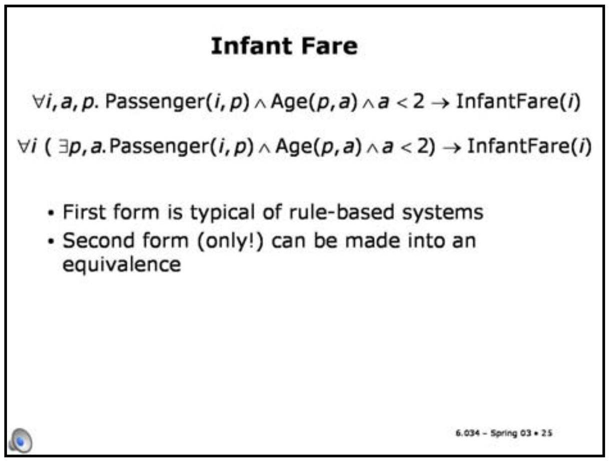
$\forall i\ (\ \exists p, a. \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2) \rightarrow \text{InfantFare}(i)$

6.034 – Spring 03 • 24

Slide 10.7.24
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

```
all i. (exists a, p. Passenger(i,p) ^ Age(p,a) ^ a < 2)
                                            -> Infant_Fare(i)
```

Slide 10.7.25
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.



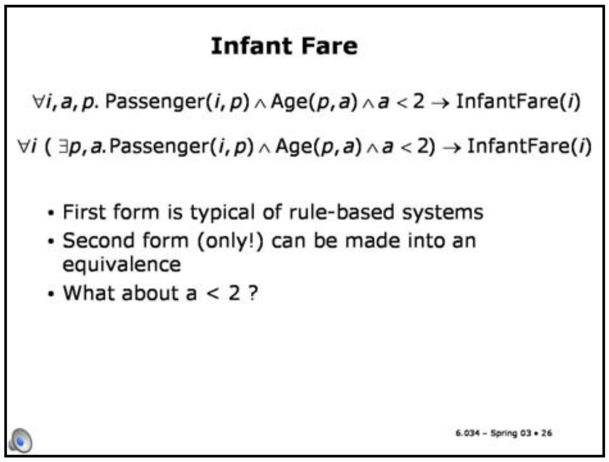
Infant Fare

$\forall i, a, p.\ \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{InfantFare}(i)$

$\forall i\ (\ \exists p, a. \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2) \rightarrow \text{InfantFare}(i)$

- First form is typical of rule-based systems
- Second form (only!) can be made into an equivalence

6.034 – Spring 03 • 25



Infant Fare

$\forall i, a, p.\ \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2 \rightarrow \text{InfantFare}(i)$

$\forall i\ (\ \exists p, a. \text{Passenger}(i, p) \wedge \text{Age}(p, a) \wedge a < 2) \rightarrow \text{InfantFare}(i)$

- First form is typical of rule-based systems
- Second form (only!) can be made into an equivalence
- What about a < 2 ?

6.034 – Spring 03 • 26

Slide 10.7.26
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.

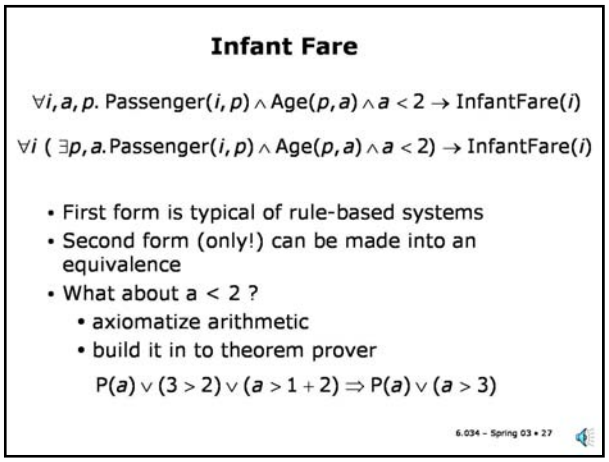
Slide 10.7.27
In this domain we'll need addition and subtraction and greater-than. One strategy would be to axiomatize them in logic, but that's usually wildly inefficient. Most systems for doing practical logical inference include built-in arithmetic predicates that will automatically evaluate themselves if their arguments are instantiated. So if, during the course of a resolution proof, you had a clause of the form

```
P(a) v (3 < 2) v (a > 1 + 2)
```

it would automatically simplify to

```
P(a) v (a > 3)
```

.



Infant Fare

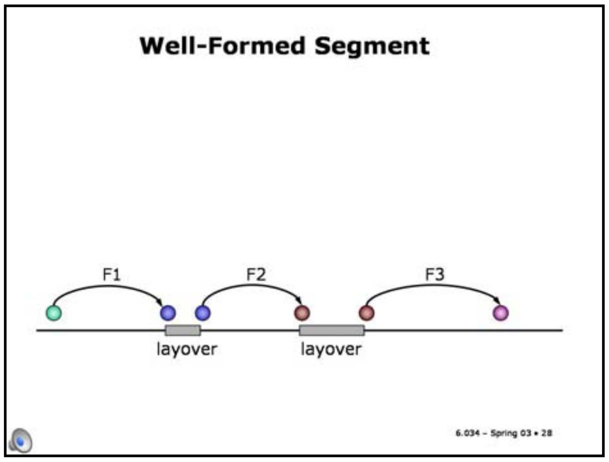
$\forall i, a, p.\ \text{Passenger}(i,p) \wedge \text{Age}(p,a) \wedge a < 2 \rightarrow \text{InfantFare}(i)$

$\forall i\ (\exists p, a.\ \text{Passenger}(i,p) \wedge \text{Age}(p,a) \wedge a < 2) \rightarrow \text{InfantFare}(i)$

- First form is typical of rule-based systems
- Second form (only!) can be made into an equivalence
- What about a < 2 ?
  - axiomatize arithmetic
  - build it in to theorem prover

$P(a) \vee (3 > 2) \vee (a > 1 + 2) \Rightarrow P(a) \vee (a > 3)$

6.034 – Spring 03 • 27



Well-Formed Segment

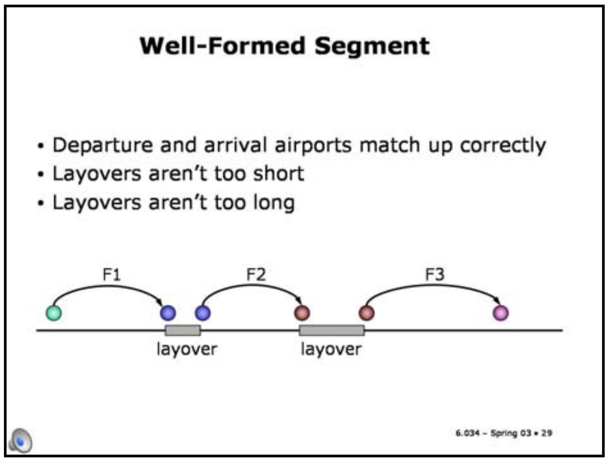
6.034 – Spring 03 • 28

Slide 10.7.28
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 10.7.29
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.

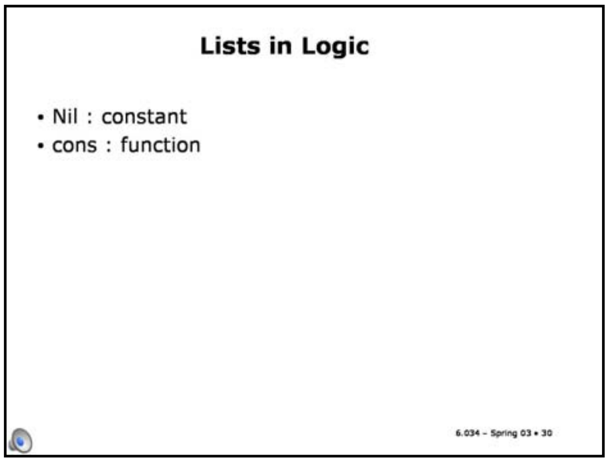


Well-Formed Segment

- Departure and arrival airports match up correctly
- Layovers aren't too short
- Layovers aren't too long

F1
F2
F3
layover
layover

6.034 – Spring 03 • 29



Lists in Logic

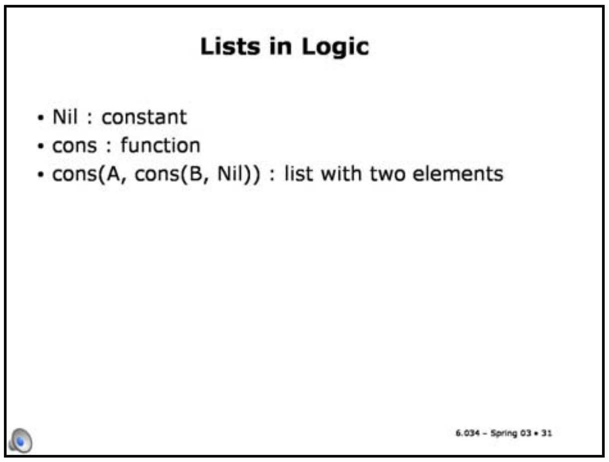
- Nil : constant
- cons : function

6.034 – Spring 03 • 30

Slide 10.7.30
In the list of relations for the domain, we included a constant Nil and a function cons, without explanation. Here's the explanation.

Slide 10.7.31
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).

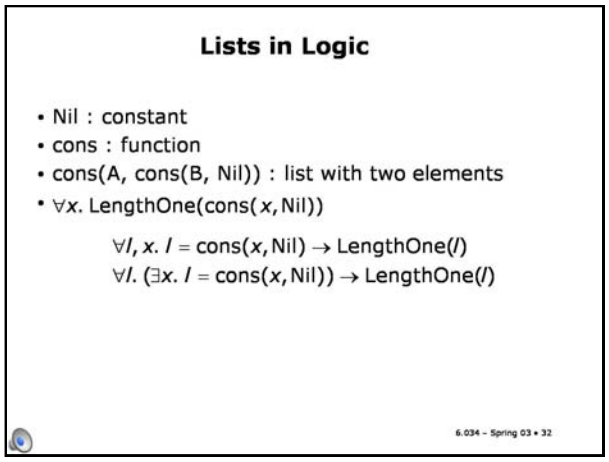
So cons(A,cons(B,Nil)) is a list with two elements, A and B.



Lists in Logic

- Nil : constant
- cons : function
- cons(A, cons(B, Nil)) : list with two elements

6.034 – Spring 03 • 31



Lists in Logic

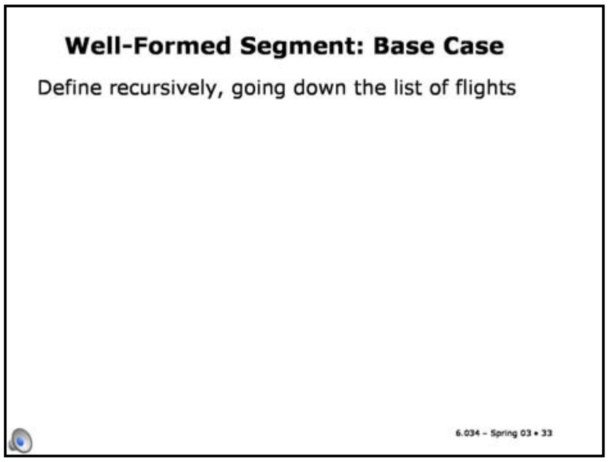
- Nil : constant
- cons : function
- cons(A, cons(B, Nil)) : list with two elements
- $\forall x.$ LengthOne(cons(x, Nil))

$\forall l, x.\ l = \text{cons}(x, \text{Nil}) \rightarrow \text{LengthOne}(l)$
$\forall l.\ (\exists x.\ l = \text{cons}(x, \text{Nil})) \rightarrow \text{LengthOne}(l)$

6.034 – Spring 03 • 32

Slide 10.7.32
We can also use the power of unification to specify conditions to assertions. So we can write **for all 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.

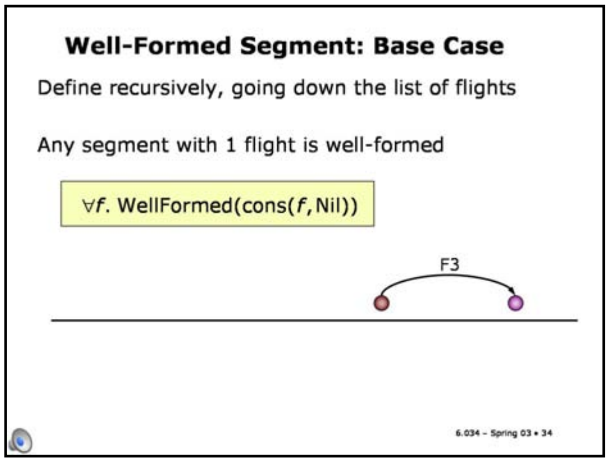
Slide 10.7.33
Now that we know how to do some things with lists, we'll go back to the problem of ensuring that a flight segment is well-formed. This is basically a condition on all the layovers in the segment, so we'll have to run down the list making sure it's all okay.



Well-Formed Segment: Base Case

Define recursively, going down the list of flights

6.034 – Spring 03 • 33



Well-Formed Segment: Base Case

Define recursively, going down the list of flights

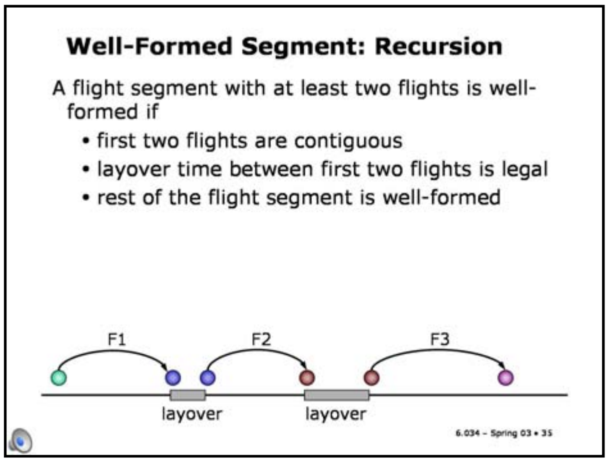
Any segment with 1 flight is well-formed

$\boxed{\forall f.\ \text{WellFormed}(\text{cons}(f, \text{Nil}))}$

6.034 – Spring 03 • 34

Slide 10.7.34
We can start with a simple case. If the flight segment has one flight, then it's well-formed. We can write this as **all f. WellFormed(cons(f, Nil))**.

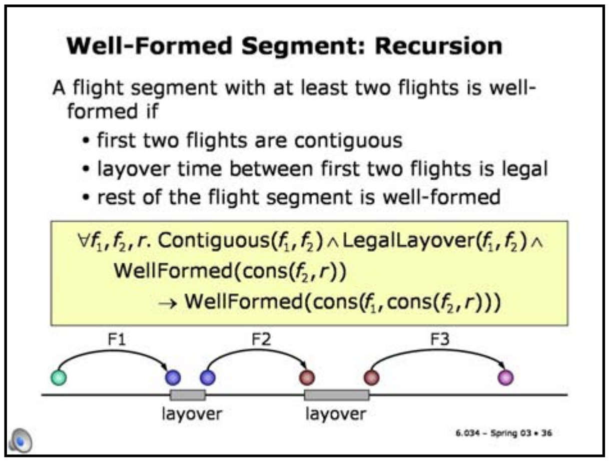
Slide 10.7.35
Now, let's do the hard case. We can say that a flight segment with more than one flight is well-formed if the first two flights are contiguous (end and start in the same airport), the layover time between the first two flights is legal, and the rest of the flight segment is well-formed.



Well-Formed Segment: Recursion

A flight segment with at least two flights is well-formed if
- first two flights are contiguous
- layover time between first two flights is legal
- rest of the flight segment is well-formed

F1
F2
F3
layover
layover
6.034 - Spring 03 • 35



Well-Formed Segment: Recursion

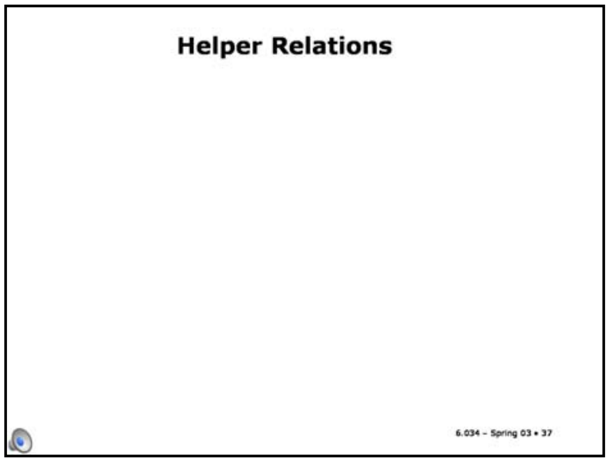
A flight segment with at least two flights is well-formed if
- first two flights are contiguous
- layover time between first two flights is legal
- rest of the flight segment is well-formed

$$\forall f_1, f_2, r.\ \text{Contiguous}(f_1, f_2) \wedge \text{LegalLayover}(f_1, f_2) \wedge \text{WellFormed}(\text{cons}(f_2, r)) \rightarrow \text{WellFormed}(\text{cons}(f_1, \text{cons}(f_2, r)))$$

Slide 10.7.36
In logic, that becomes

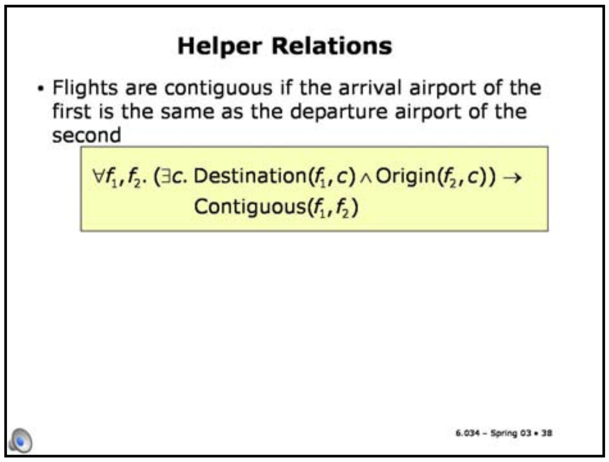
```
all f1, f2, r. Contiguous(f1, f2) ^ LegalLayover(f1, f2) ^
       WellFormed(cons(f2, r)) -> WellFormed(cons(f1, cons(f2,r)))
```

Slide 10.7.37
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 divide our problem into conquerable sub-parts.



Helper Relations

6.034 - Spring 03 • 37



Helper Relations

- Flights are contiguous if the arrival airport of the first is the same as the departure airport of the second

$$\forall f_1, f_2.\ (\exists c.\ \text{Destination}(f_1, c) \wedge \text{Origin}(f_2, c)) \rightarrow \text{Contiguous}(f_1, f_2)$$

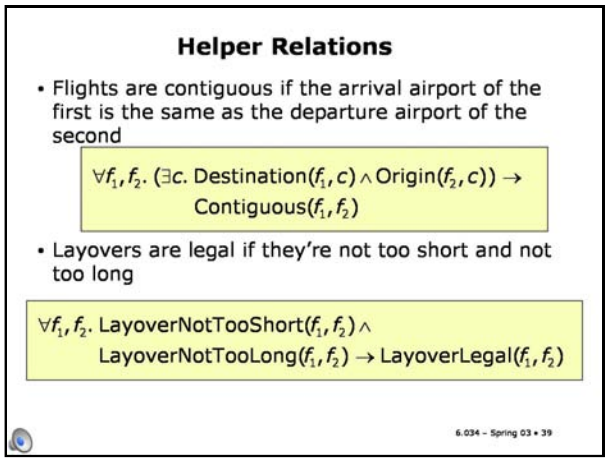
6.034 - Spring 03 • 38

Slide 10.7.38
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) ^ Origin(f2, c))
                  -> Contiguous(f1, f2)
```

Slide 10.7.39
Now, what makes layovers legal? They have to be not too short and not too long.

```
all f1, f2.
    LayoverNotTooShort(f1, f2) ^ LayoverNotTooLong(f1, f2) ->
        LayoverLegal(f1, f2)
```



Helper Relations

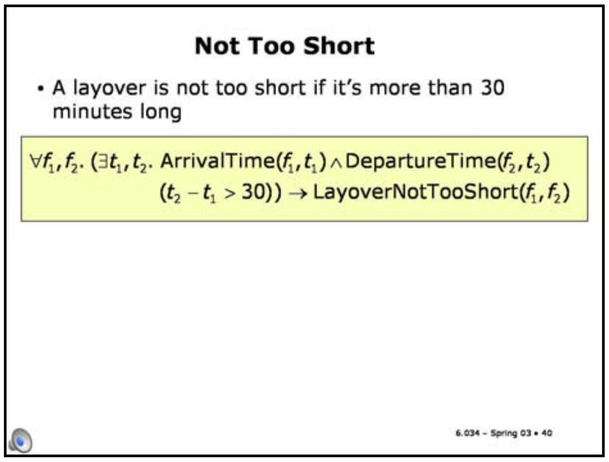
- Flights are contiguous if the arrival airport of the first is the same as the departure airport of the second

$$\forall f_1, f_2. (\exists c. \text{Destination}(f_1, c) \wedge \text{Origin}(f_2, c)) \rightarrow \text{Contiguous}(f_1, f_2)$$

- Layovers are legal if they're not too short and not too long

$$\forall f_1, f_2. \text{LayoverNotTooShort}(f_1, f_2) \wedge \text{LayoverNotTooLong}(f_1, f_2) \rightarrow \text{LayoverLegal}(f_1, f_2)$$

6.034 – Spring 03 • 39



Not Too Short

- A layover is not too short if it's more than 30 minutes long

$$\forall f_1, f_2. (\exists t_1, t_2. \text{ArrivalTime}(f_1, t_1) \wedge \text{DepartureTime}(f_2, t_2) (t_2 - t_1 > 30)) \rightarrow \text{LayoverNotTooShort}(f_1, f_2)$$

6.034 – Spring 03 • 40

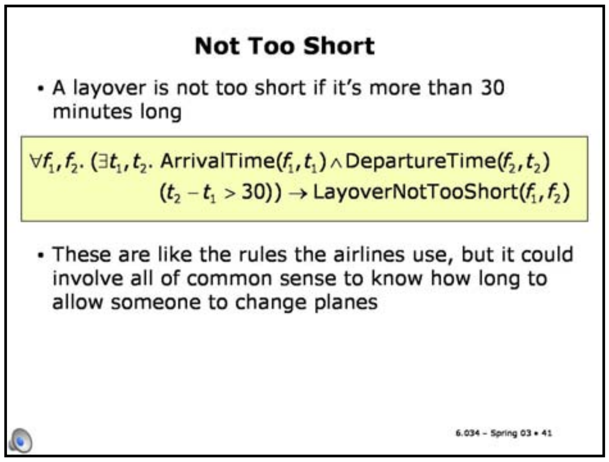
Slide 10.7.40
Let's say that passengers need at least 30 minutes to change planes.

That comes out fairly straightforwardly as

```
all f1, f2. (exists t1, t2.
    Arrival_Time(f1, t1) ^ Departure_Time(f2, t2) ^ (t2 - t1 >
30)) ->
                     Layover_Not_Too_Short(f1, f2)
```

Slide 10.7.41
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 base 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.



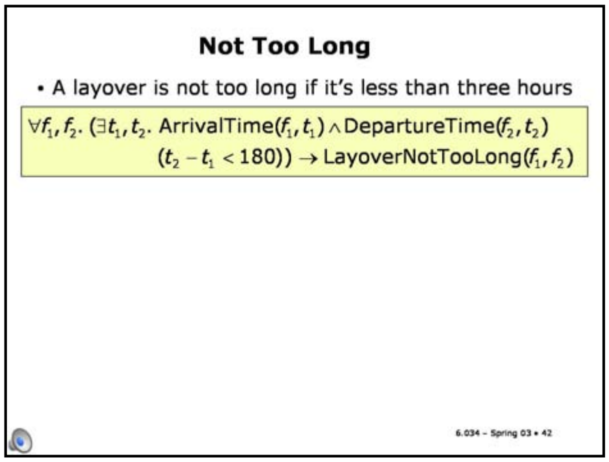
Not Too Short

- A layover is not too short if it's more than 30 minutes long

$$\forall f_1, f_2. (\exists t_1, t_2. \text{ArrivalTime}(f_1, t_1) \wedge \text{DepartureTime}(f_2, t_2) (t_2 - t_1 > 30)) \rightarrow \text{LayoverNotTooShort}(f_1, f_2)$$

- These are like the rules the airlines use, but it could involve all of common sense to know how long to allow someone to change planes

6.034 – Spring 03 • 41



Not Too Long

- A layover is not too long if it's less than three hours

$$\forall f_1, f_2. (\exists t_1, t_2. \text{ArrivalTime}(f_1, t_1) \wedge \text{DepartureTime}(f_2, t_2) (t_2 - t_1 < 180)) \rightarrow \text{LayoverNotTooLong}(f_1, f_2)$$

6.034 – Spring 03 • 42

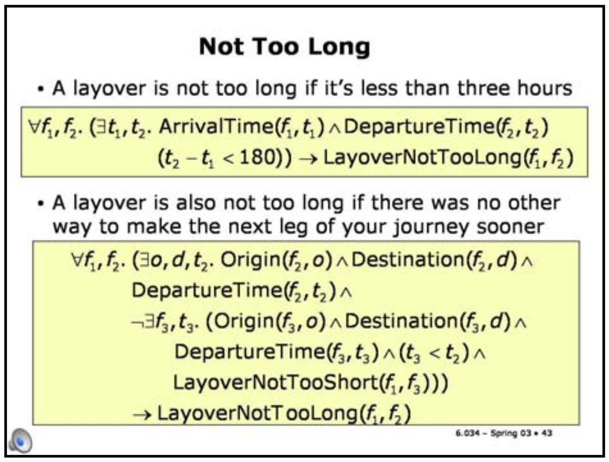
Slide 10.7.42
We'll continue in our more circumscribed setting, to address the question of what makes a layover not be too long. This will have an easy case and a hard case. The easy case is that a layover is not too long if it's less than three hours:

```
all f1, f2. (exists t1, t2.
    ArrivalTime(f1, t1) ^ DepartureTime(f2, t2) ^
    (t2 - t1 < 180)) -> LayoverNotTooLong(f1, f2)
```

Slide 10.7.43
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).

```
all f1, f2 (exists o, d, t2.
      Origin(f2, o) ^ Destination(f2, d) ^ DepartureTime(f2,t2) ^
            ~ exists f3, t3. ( Origin(f3, o) ^ Destination(f3, d)
^
                           DepartureTime(f3,t3) ^ (t3 < t2)
                         ^  LayoverNotTooShort(f1, f3))) ->
                LayoverNotTooLong(f1, f2)
```

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



Not Too Long

- A layover is not too long if it's less than three hours

$$\forall f_1, f_2.\ (\exists t_1, t_2.\ \text{ArrivalTime}(f_1, t_1) \wedge \text{DepartureTime}(f_2, t_2)\ (t_2 - t_1 < 180)) \rightarrow \text{LayoverNotTooLong}(f_1, f_2)$$

- A layover is also not too long if there was no other way to make the next leg of your journey sooner

$$\forall f_1, f_2.\ (\exists o, d, t_2.\ \text{Origin}(f_2, o) \wedge \text{Destination}(f_2, d) \wedge \text{DepartureTime}(f_2, t_2) \wedge \neg\exists f_3, t_3.\ (\text{Origin}(f_3, o) \wedge \text{Destination}(f_3, d) \wedge \text{DepartureTime}(f_3, t_3) \wedge (t_3 < t_2) \wedge \text{LayoverNotTooShort}(f_1, f_3))) \rightarrow \text{LayoverNotTooLong}(f_1, f_2)$$

6.034 - Spring 03 • 43

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.