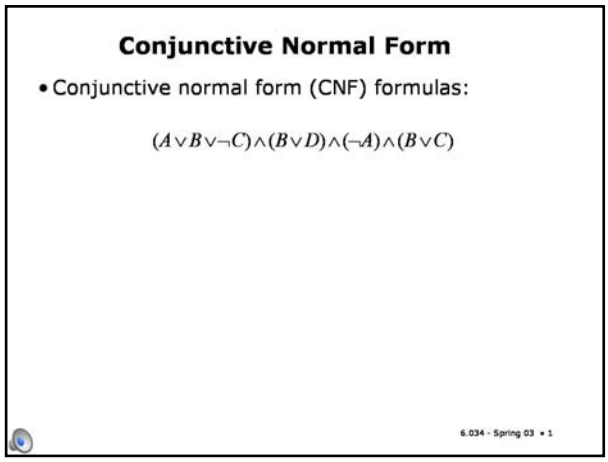
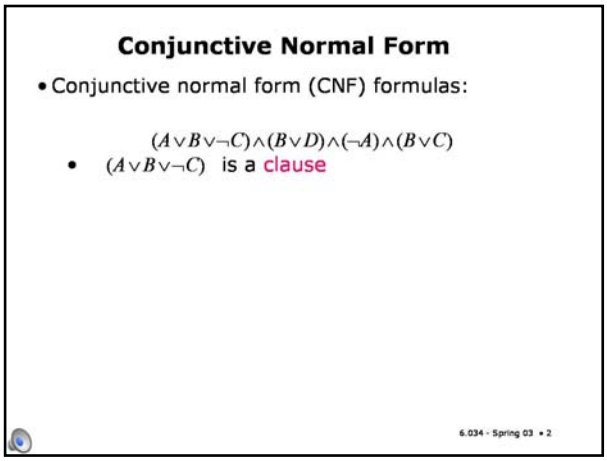
6.034 Notes: Section 10.1

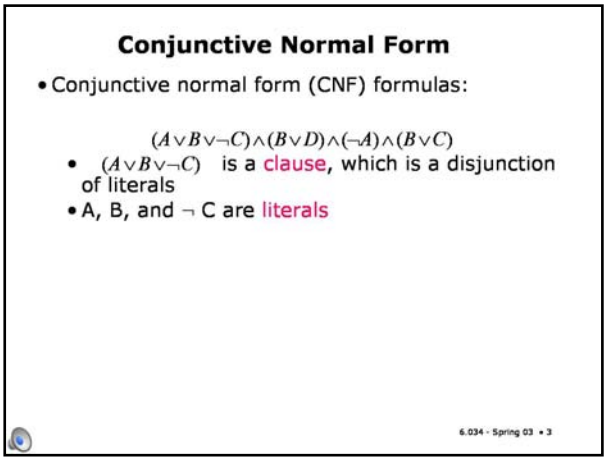
Slide 10.1.1
A sentence written in conjunctive normal form looks like ((A or B or not C) and (B or D) and (not A) and (B or C)).

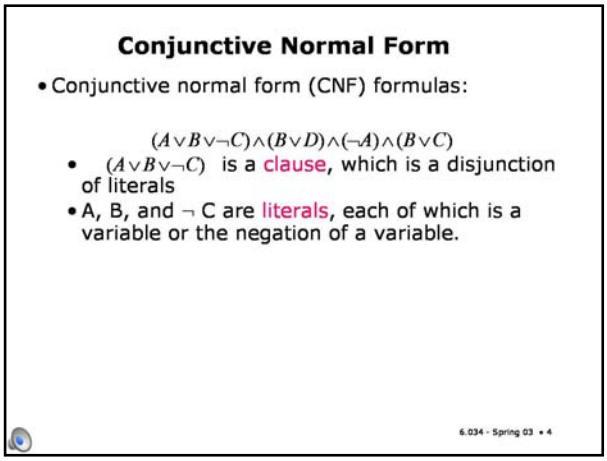




Slide 10.1.2
Its outermost structure is a conjunction. It's a conjunction of multiple units. These units are called "clauses."

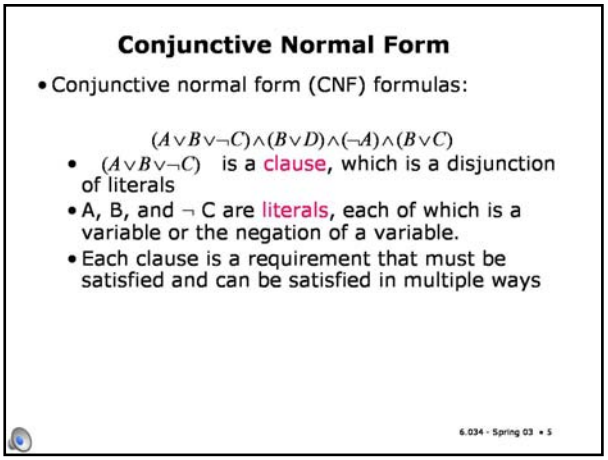
Slide 10.1.3
A clause is the disjunction of many things. The units that make up a clause are called literals.

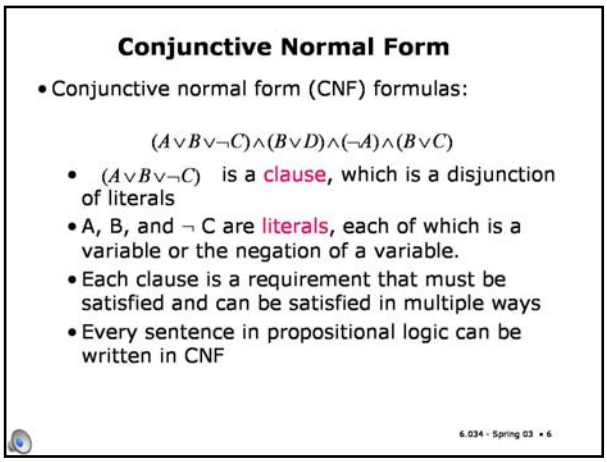




Slide 10.1.4
And a literal is either a variable or the negation of a variable.

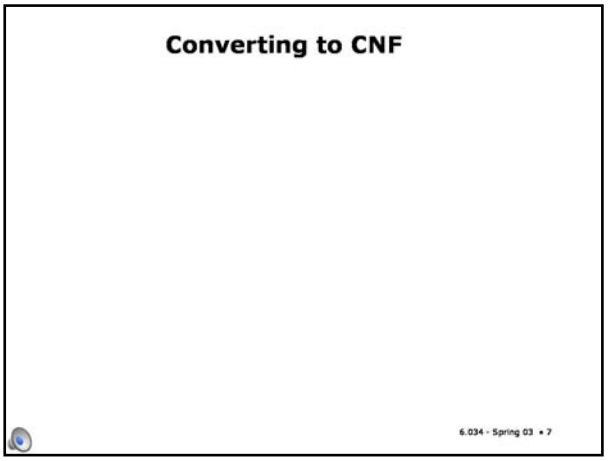
Slide 10.1.5
So you get an expression where the negations are pushed in as tightly as possible, then you have ors, then you have ands. This is like saying that every assignment has to meet each of a set of requirements. You can think of each clause as a requirement. So somehow, the first clause has to be satisfied, and it has different ways that it can be satisfied, and the second one has to be satisfied, and the third one has to be satisfied, and so on.

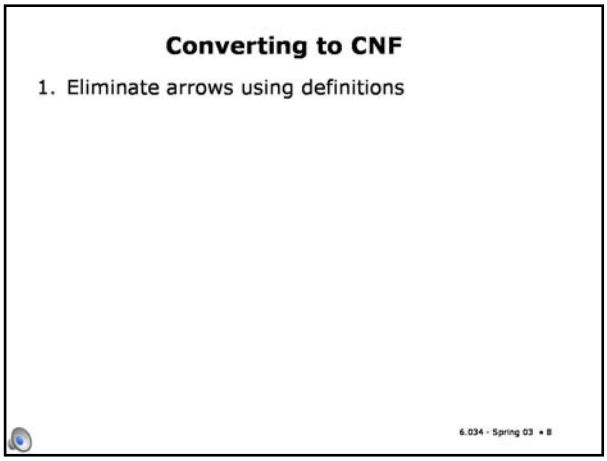




Slide 10.1.6
You can take any sentence in propositional logic and write it in conjunctive normal form.

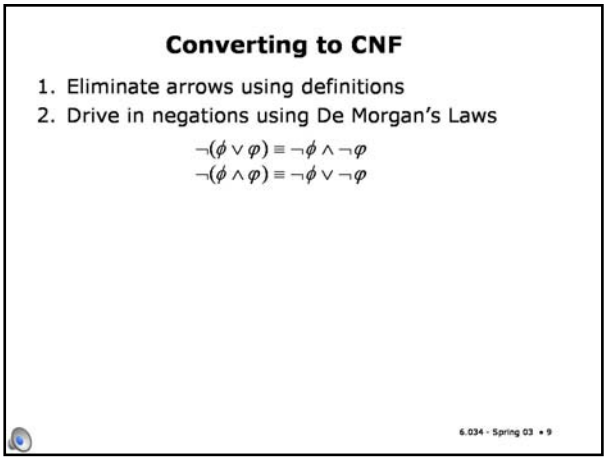
Slide 10.1.7
Here's the procedure for converting sentences to conjunctive normal form.

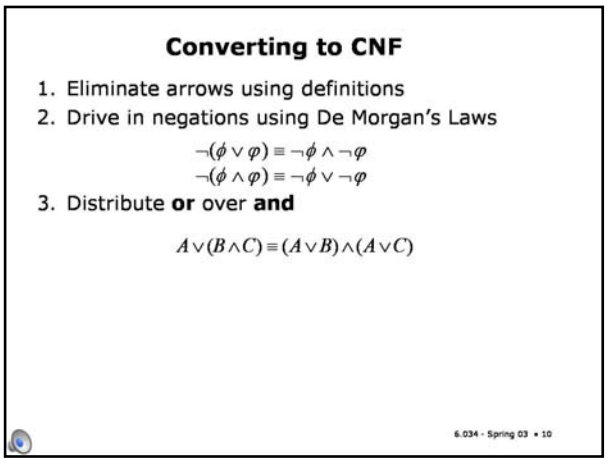




Slide 10.1.8
The first step is to eliminate single and double arrows using their definitions.

Slide 10.1.9
The next step is to drive in negation. We do it using DeMorgan's Laws. You might have seen them in a digital logic class. Not (phi or psi) is equivalent to (not phi and not psi). And, Not (phi and psi) is equivalent to (not phi or not psi). So if you have a negation on the outside, you can push it in and change the connective from and to or, or from or to and.

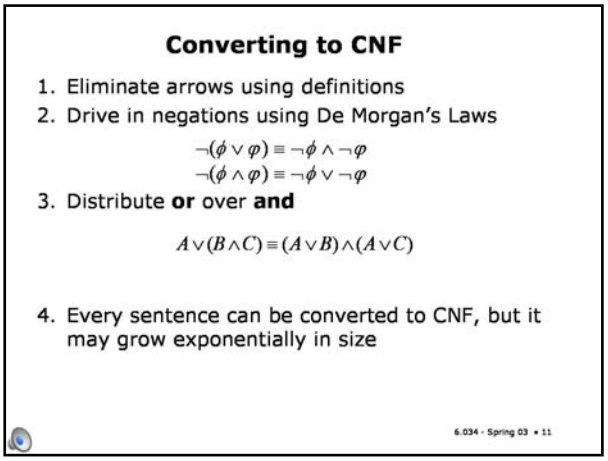


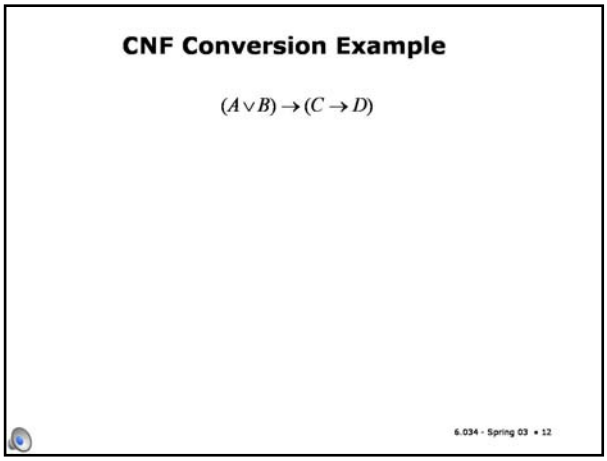


Slide 10.1.10
The third step is to distribute or over and. That is, if we have (A or (B and C)) we can rewrite it as (A or B) and (A or C). You can prove to yourself, using the method of truth tables, that the distribution rule (and DeMorgan's laws) are valid.

Slide 10.1.11
One problem with conjunctive normal form is that, although you can convert any sentence to conjunctive normal form, you might do it at the price of an exponential increase in the size of the expression. Because if you have A and B and C OR D and E and F, you end up making the cross-product of all of those things.

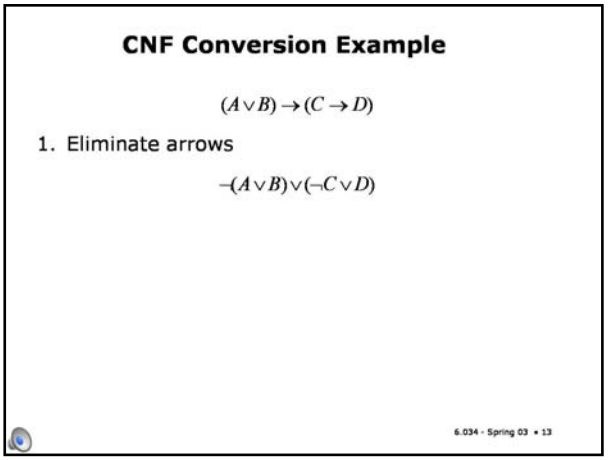
For now, we'll think about satisfiability problems, which are generally fairly efficiently converted into CNF.

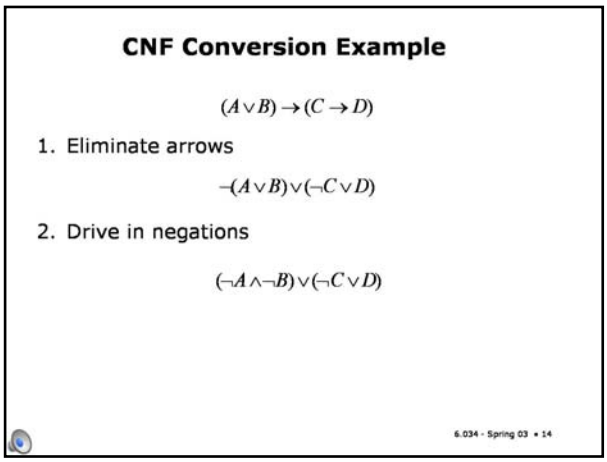




Slide 10.1.12
Here's an example of converting a sentence to CNF.

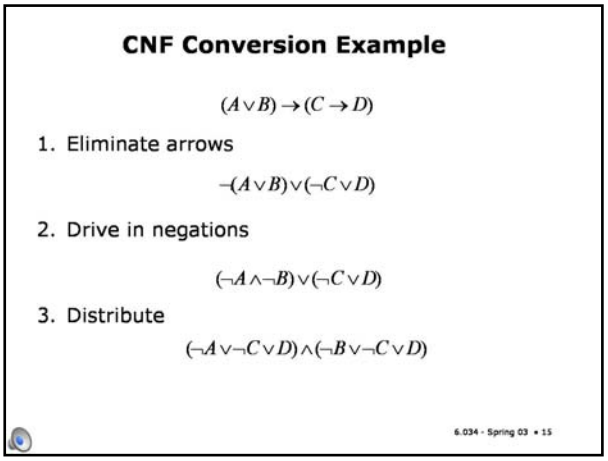
Slide 10.1.13
First we get rid of both arrows, using the rule that says "A implies B" is equivalent to "not A or B".





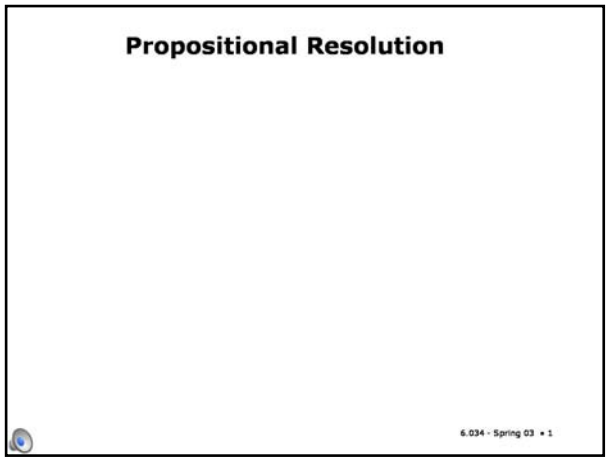
Slide 10.1.14
Then we drive in the negation using deMorgan's law.

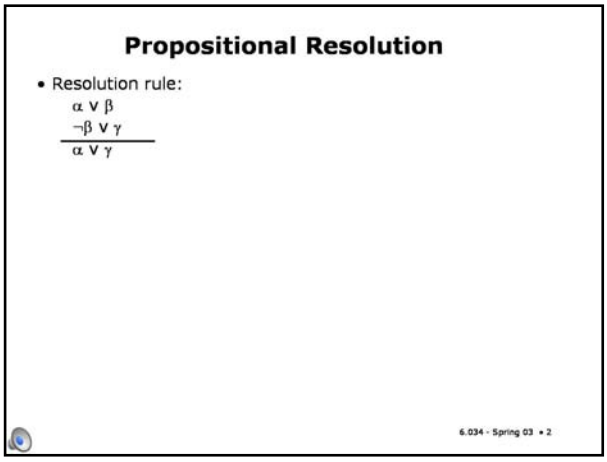
Slide 10.1.15
Finally, we distribute or over and to get the final CNF expression.



6.034 Notes: Section 10.2

Slide 10.2.1
We have talked a little bit about proof, with the idea that you write down some axioms -- statements that you're given -- and then you try to derive something from them. And we've all had practice doing that in high school geometry and we've talked a little bit about natural deduction. So what we're going to talk about now is resolution. Which is the way that pretty much every modern automated theorem-prover is implemented. It seems to be the best way for computers to think about proving things.

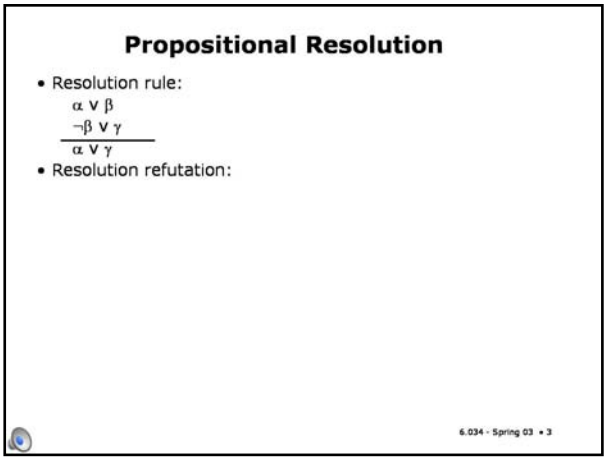


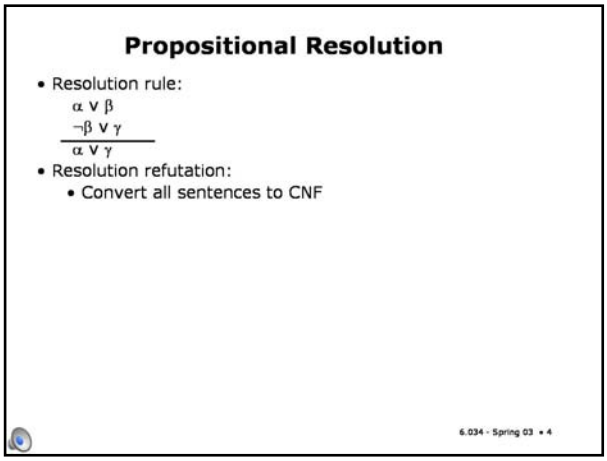


Slide 10.2.2
So here's the **resolution** inference rule, in the propositional case. It says that if you know "alpha or beta", and you know "not beta or gamma", then you're allowed to conclude "alpha or gamma".

Remember from when we looked at inference rules before that these Greek letters are meta-variables. They can stand for big chunks of propositional logic, as long as the parts match up in the right way. So if you know something of the form "alpha or beta", and you also know that "not beta or gamma", then you can conclude "alpha or gamma".

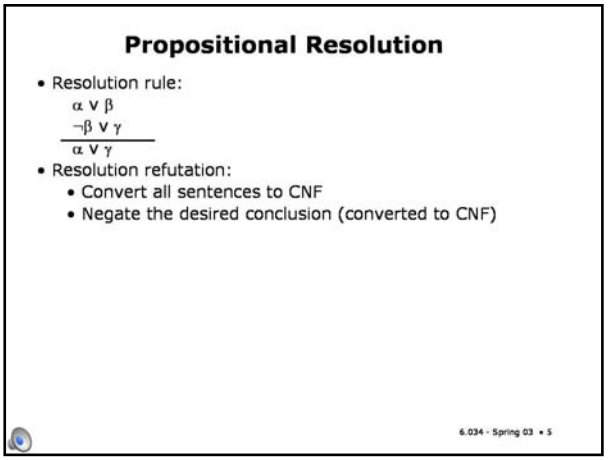
Slide 10.2.3
It turns out that this one rule is all you need to prove anything in propositional logic. At least, to prove that a set of sentences is not satisfiable. So, let's see how this is going to work. There's a proof strategy called **resolution refutation**, with three steps. It goes like this.

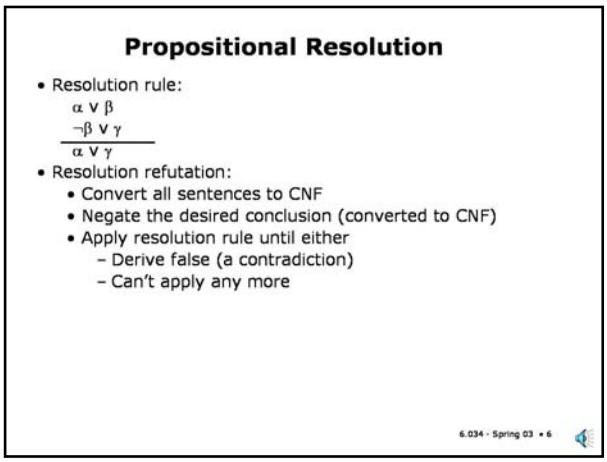




Slide 10.2.4
First, you convert all of your sentences to conjunctive normal form. You already know how to do this! Then, you write each clause down as a premise or given in your proof.

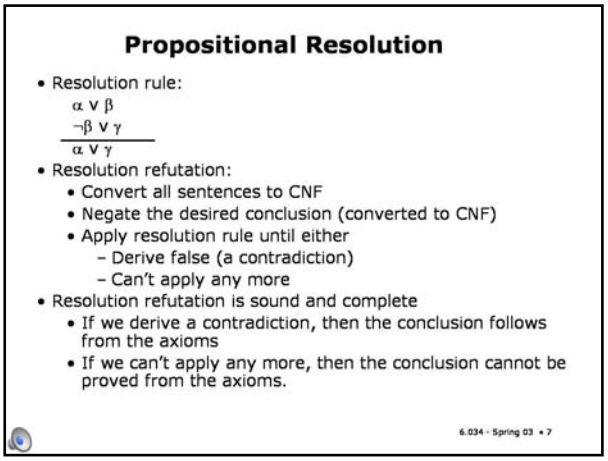
Slide 10.2.5
Then, you negate the desired conclusion -- so you have to say what you're trying to prove, but what we're going to do is essentially a proof by contradiction. You've all seen the strategy of proof by contradiction (or, if we're being fancy and Latin, reductio ad absurdum). You assert that the thing that you're trying to prove is false, and then you try to derive a contradiction. That's what we're going to do. So you negate the desired conclusion and convert that to CNF. And you add each of these clauses as a premise of your proof, as well.

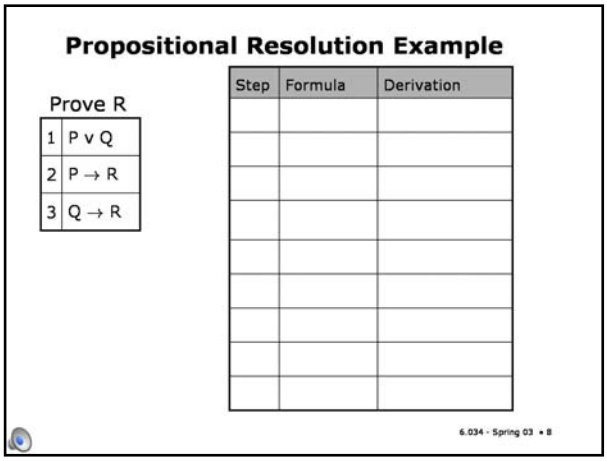




Slide 10.2.6
Now we apply the resolution rule until one of two things happens. We might derive "false", which means that the conclusion did, in fact, follow from the things that we had assumed. If you assert that the negation of the thing that you're interested in is true, and then you prove for a while and you manage to prove false, then you've succeeded in a proof by contradiction of the thing that you were trying to prove in the first place. Or, we might find ourselves in a situation where we can't apply the resolution rule any more, but we still haven't managed to derive false.

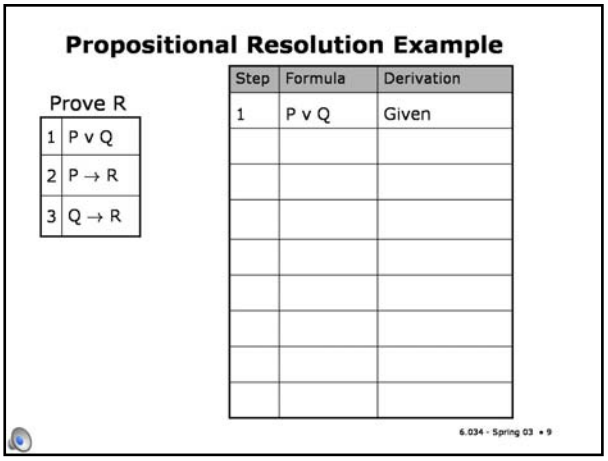
Slide 10.2.7
What if you can't apply the resolution rule anymore? Is there anything in particular that you can conclude? In fact, you can conclude that the thing that you were trying to prove can't be proved. So resolution refutation for propositional logic is a complete proof procedure. If the thing that you're trying to prove is, in fact, entailed by the things that you've assumed, then you can prove it using resolution refutation. It's guaranteed that you'll always either prove false, or run out of possible steps. It's complete, because it always generates an answer. Furthermore, the process is sound: the answer is always correct.

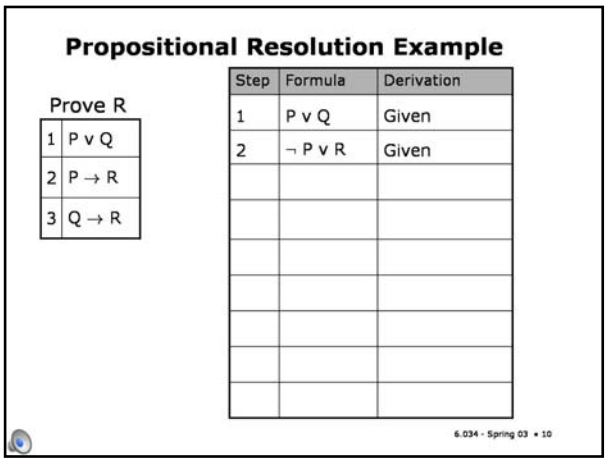




Slide 10.2.8
So let's just do a proof. Let's say I'm given "P or Q", "P implies R" and "Q implies R". I would like to conclude R from these three axioms. I'll use the word "axiom" just to mean things that are given to me right at the moment.

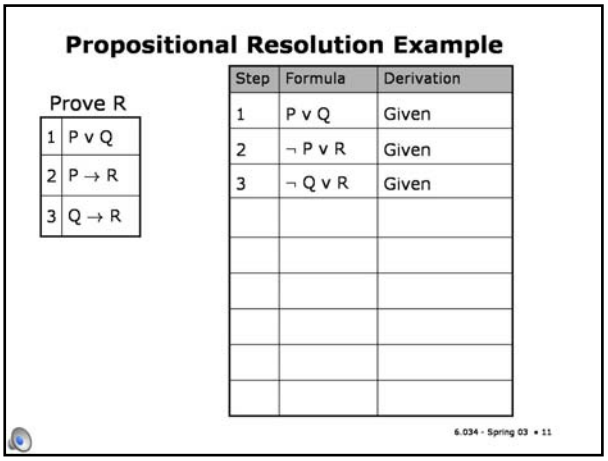
Slide 10.2.9
We start by converting this first sentence into conjunctive normal form. We don't actually have to do anything. It's already in the right form.

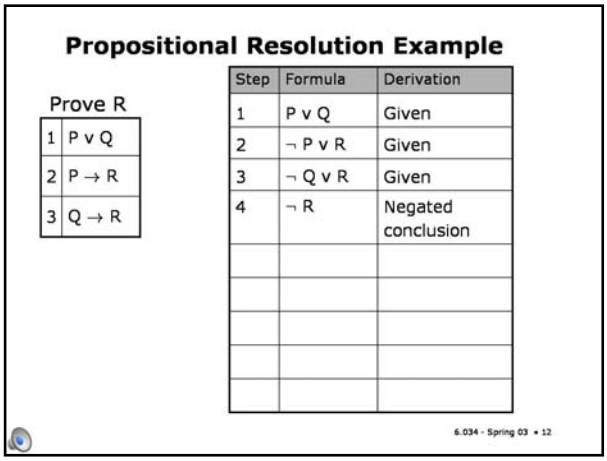




Slide 10.2.10
Now, "P implies R" turns into "not P or R".

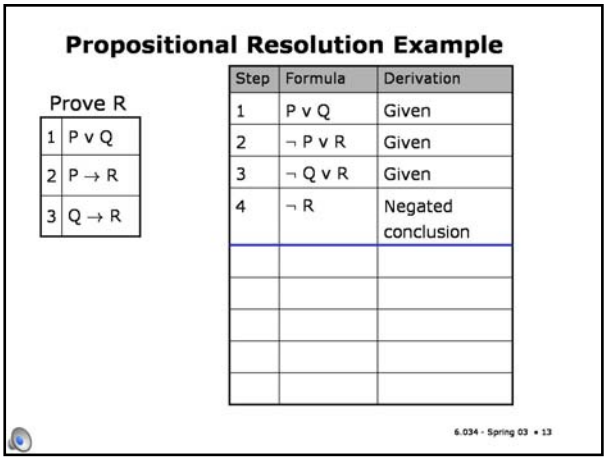
Slide 10.2.11
Similarly, "Q implies R" turns into "not Q or R"

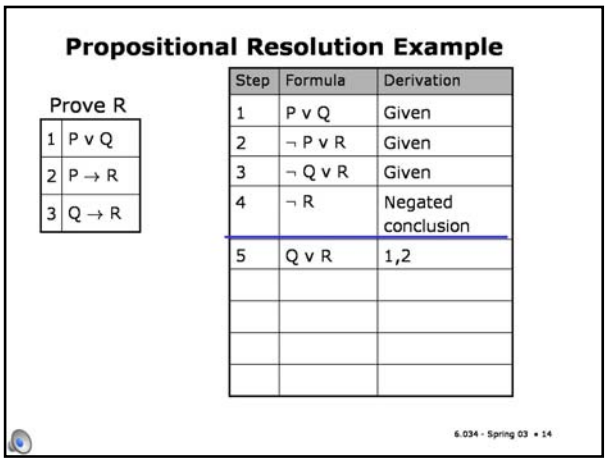




Slide 10.2.12
Now we want to add one more thing to our list of given statements. What's it going to be? Not R. Right? We're going to assert the negation of the thing we're trying to prove. We'd like to prove that R follows from these things. But what we're going to do instead is say not R, and now we're trying to prove false. And if we manage to prove false, then we will have a proof that R is entailed by the assumptions.

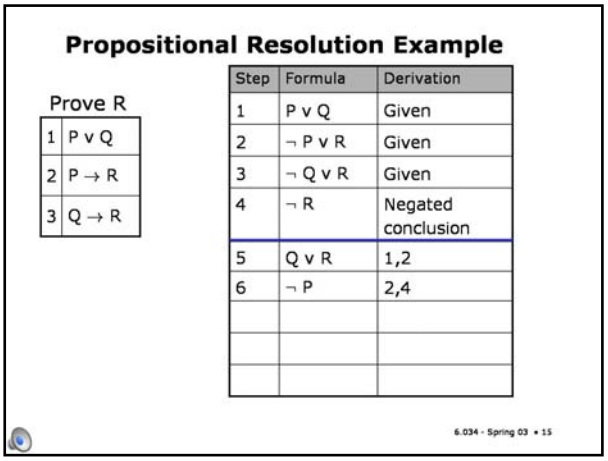
Slide 10.2.13
We'll draw a blue line just to divide the assumptions from the proof steps. And now, we look for opportunities to apply the resolution rule. You can do it in any order you like (though some orders of application will result in much shorter proofs than others).

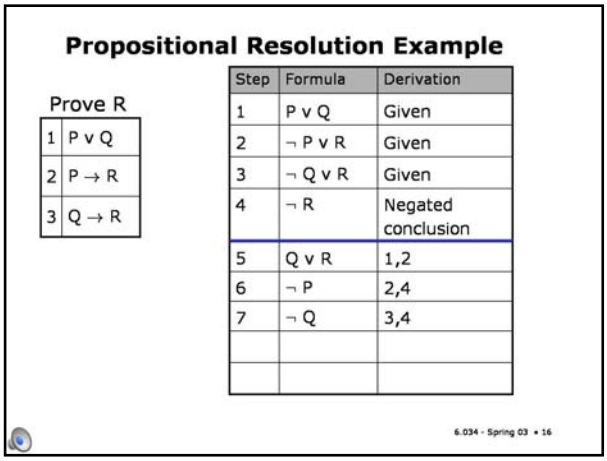




Slide 10.2.14
We can apply resolution to lines 1 and 2, and get "Q or R" by resolving away P.

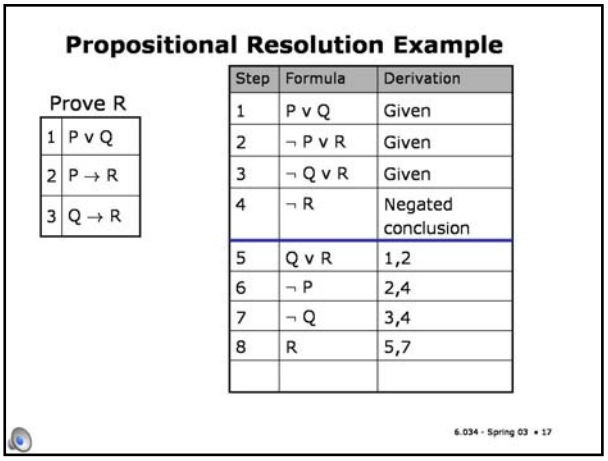
Slide 10.2.15
And we can take lines 2 and 4, resolve away R, and get "not P."

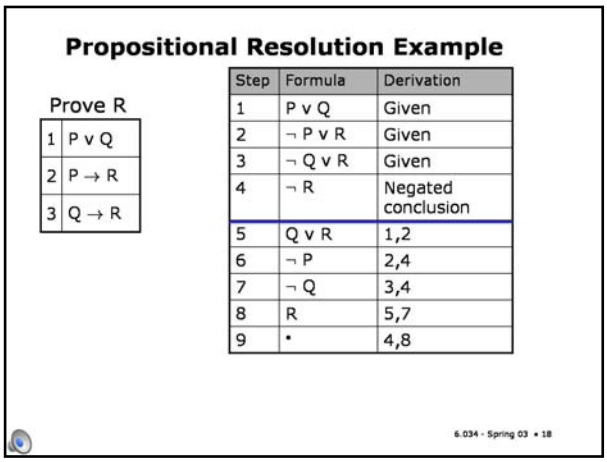




Slide 10.2.16
Similarly, we can take lines 3 and 4, resolve away R, and get "not Q".

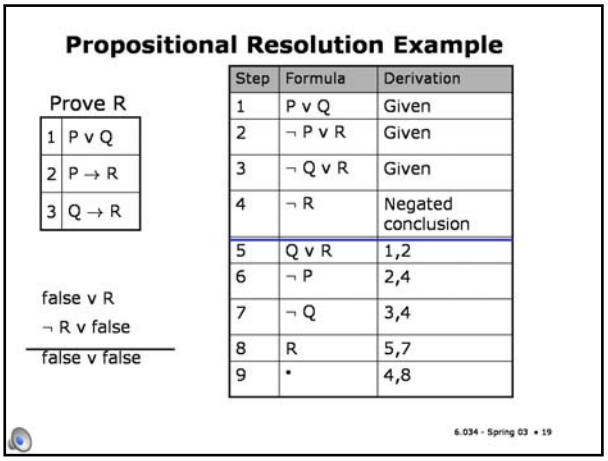
Slide 10.2.17
By resolving away Q in lines 5 and 7, we get R.

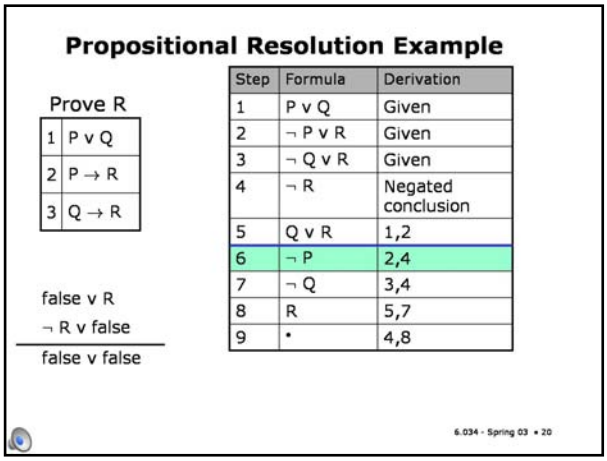




Slide 10.2.18
And finally, resolving away R in lines 4 and 8, we get the empty clause, which is false. We'll often draw this little black box to indicate that we've reached the desired contradiction.

Slide 10.2.19
How did I do this last resolution? Let's see how the resolution rule is applied to lines 4 and 8. The way to look at it is that R is really "false or R", and that "not R" is really "not R or false". (Of course, the order of the disjuncts is irrelevant, because disjunction is commutative). So, now we resolve away R, getting "false or false", which is false.

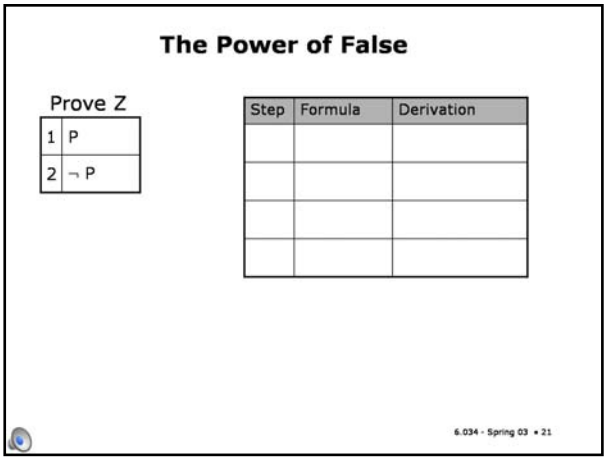


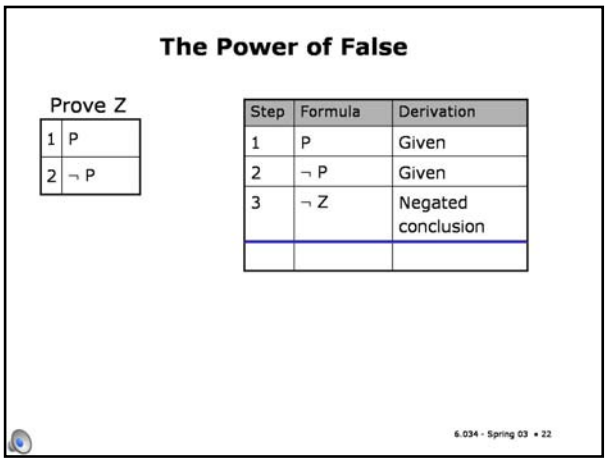


Slide 10.2.20
One of these steps is unnecessary. Which one? Line 6. It's a perfectly good proof step, but it doesn't contribute to the final conclusion, so we could have omitted it.

Slide 10.2.21
Here's a question. Does "P and not P" entail Z?

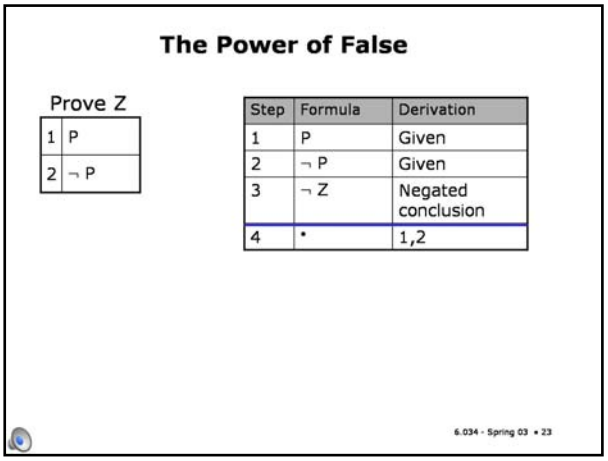
It does, and it's easy to prove using resolution refutation.

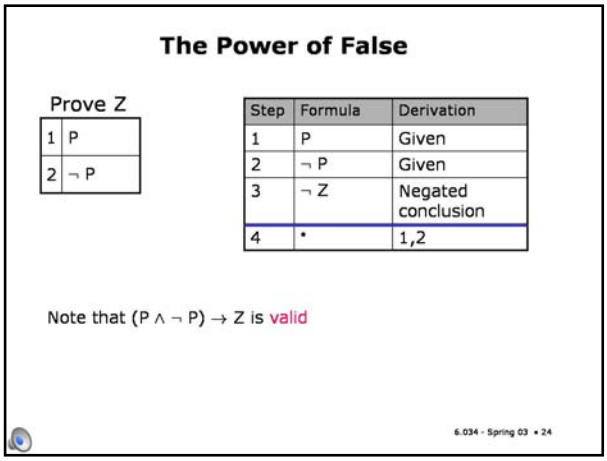




Slide 10.2.22
We start by writing down the assumptions and the negation of the conclusion.

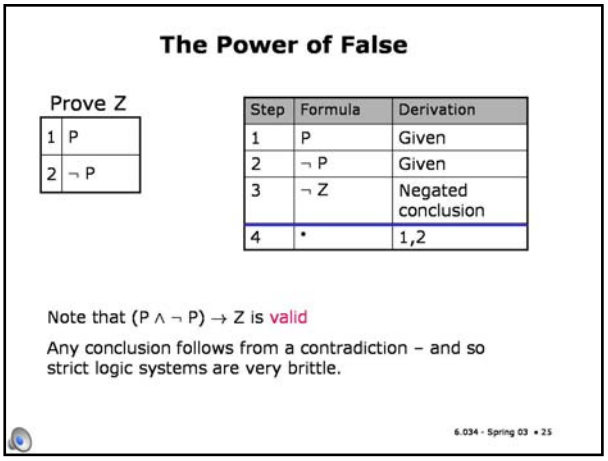
Slide 10.2.23
Then, we can resolve away P in lines 1 and 2, getting a contradiction right away.

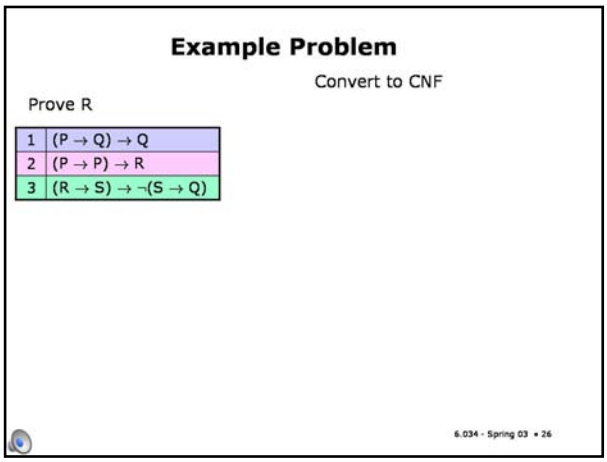




Slide 10.2.24
Because we can prove Z from "P and not P" using a sound proof procedure, then "P and not P" entails Z.

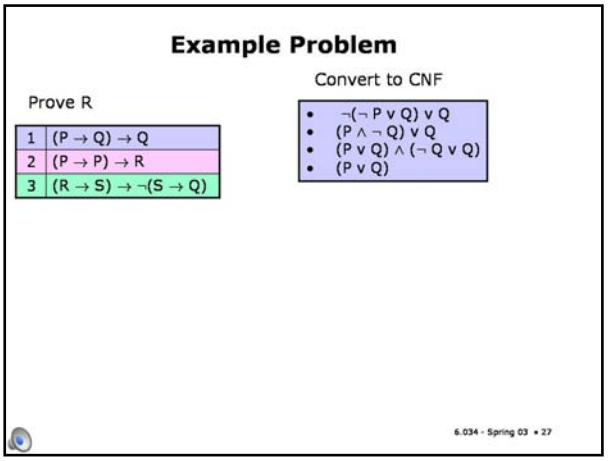
Slide 10.2.25
So, we see, again, that any conclusion follows from a contradiction. This is the property that can make logical systems quite brittle; they're not robust in the face of noise. This problem has been recently addressed in AI by a move to probabilistic reasoning methods. Unfortunately, they're out of the scope of this course.

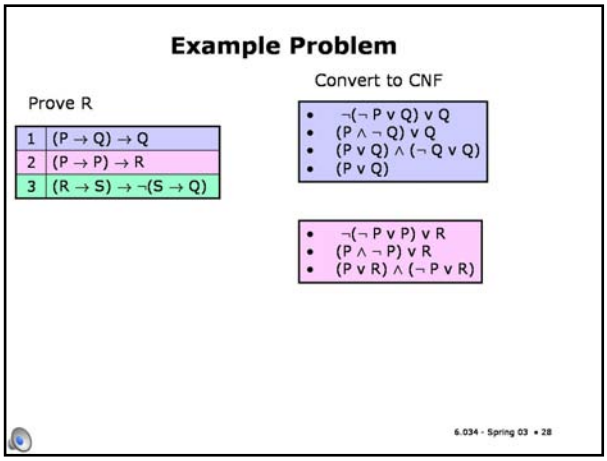




Slide 10.2.26
Here's an example problem. Stop and do the conversion into CNF before you go to the next slide.

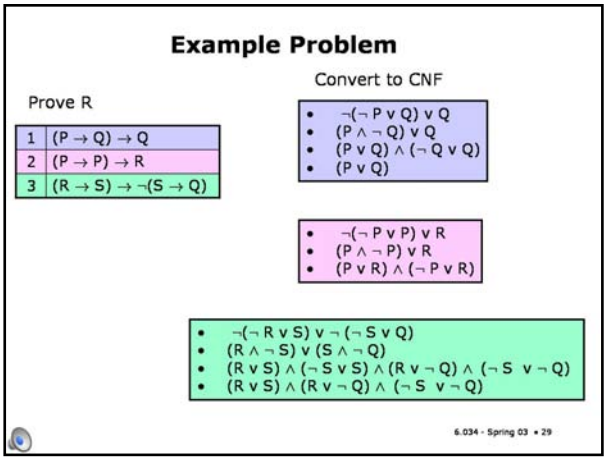
Slide 10.2.27
So, the first formula turns into "P or Q".

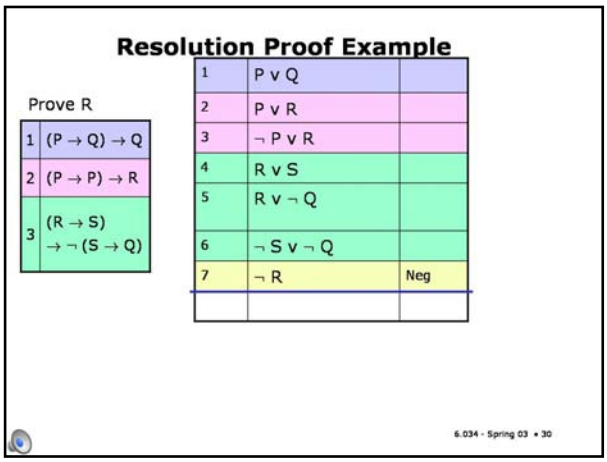




Slide 10.2.28
The second turns into ("P or R" and "not P or R"). We probably should have simplified it into "False or R" at the second step, which reduces just to R. But we'll leave it as is, for now.

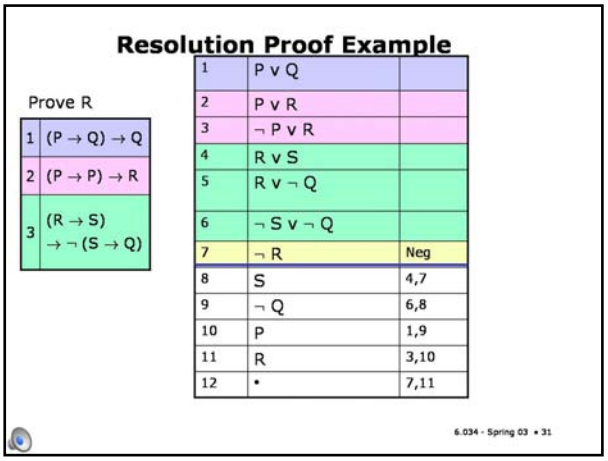
Slide 10.2.29
Finally, the last formula requires us to do a big expansion, but one of the terms is true and can be left out. So, we get "(R or S) and (R or not Q) and (not S or not Q)".

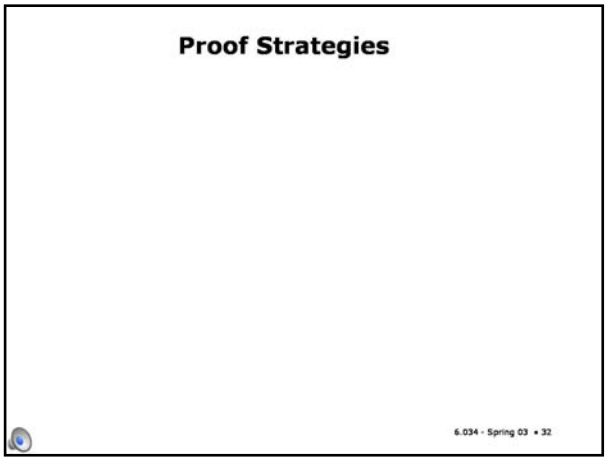




Slide 10.2.30
Now we can almost start the proof. We copy each of the clauses over here, and we add the negation of the query. Please stop and do this proof yourself before going on.

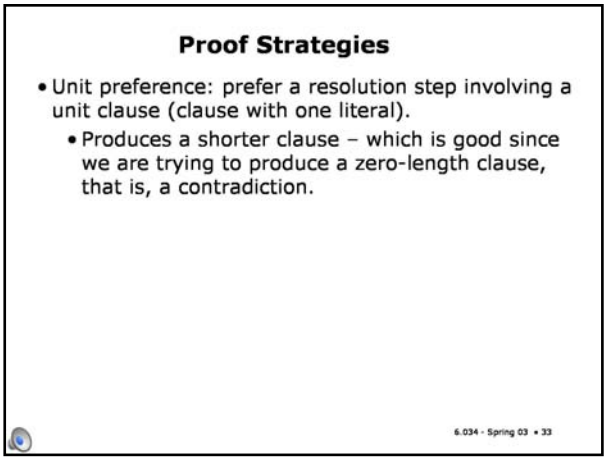
Slide 10.2.31
Here's a sample proof. It's one of a whole lot of possible proofs.

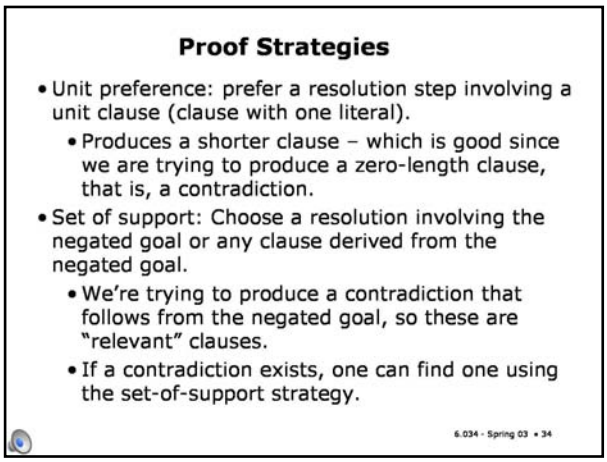




Slide 10.2.32
In choosing among all the possible proof steps that you can do at any point, there are two rules of thumb that are really important.

Slide 10.2.33
The unit preference rule says that if you can involve a clause that has only one literal in it, that's usually a good idea. It's good because you get back a shorter clause. And the shorter a clause is, the closer it is to false.

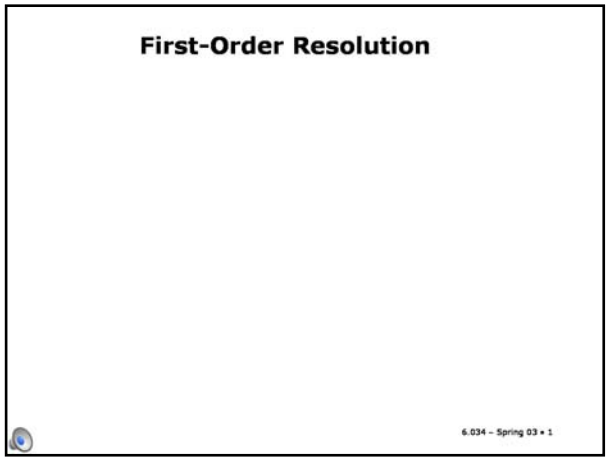


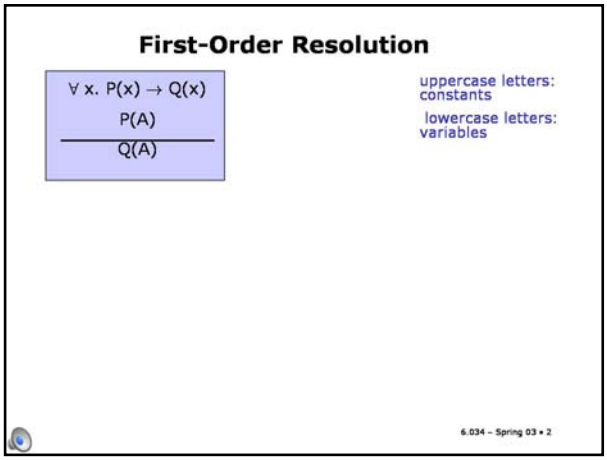


Slide 10.2.34
The set-of-support rule says you should involve the thing that you're trying to prove. It might be that you can derive conclusions all day long about the solutions to chess games and stuff from the axioms, but once you're trying to prove something about what way to run, it doesn't matter. So, to direct your "thought" processes toward deriving a contradiction, you should always involve a clause that came from the negated goal, or that was produced by the set of support rule. Adhering to the set-of-support rule will still make the resolution refutation process sound and complete.

6.034 Notes: Section 10.3

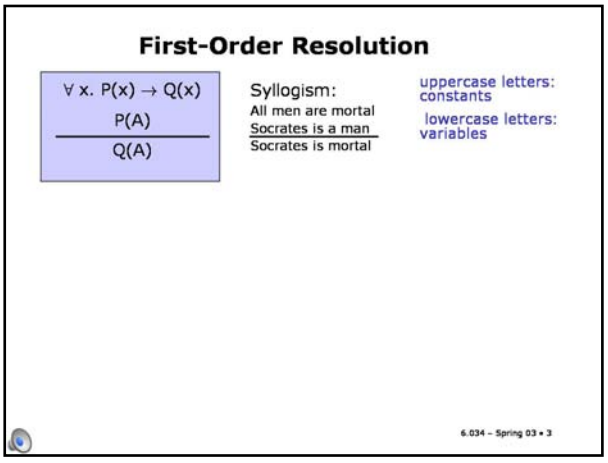
Slide 10.3.1
We are going to use resolution refutation to do proofs in first-order logic. It's a fair amount trickier than in propositional logic, though, because now we have variables to contend with.

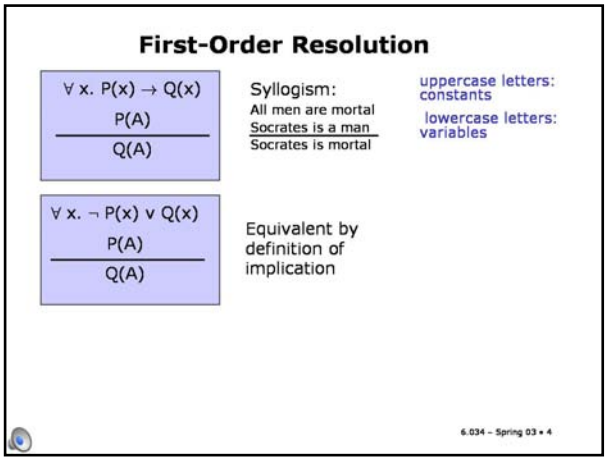




Slide 10.3.2
Let's try to get some intuition through an example. Imagine you knew "for all x, P of x implies Q of x." And let's say you also knew **P(A)**. What would you be able to conclude? **Q(A)**, right? You ought to be able to conclude **Q(A)**.

Slide 10.3.3
This is actually Aristotle's original syllogism: From "All men are mortal" and "Socrates is a man", conclude "Socrates is a mortal".

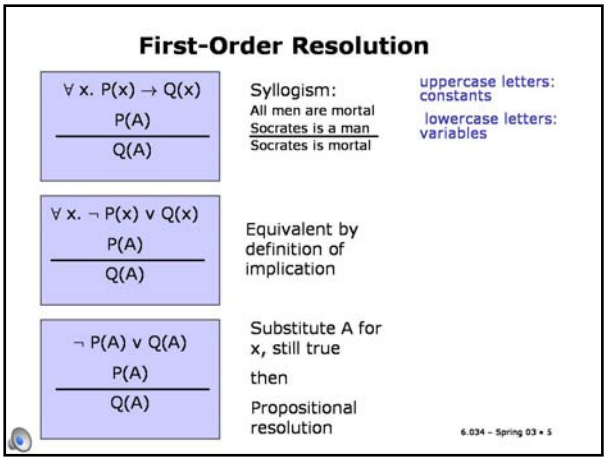


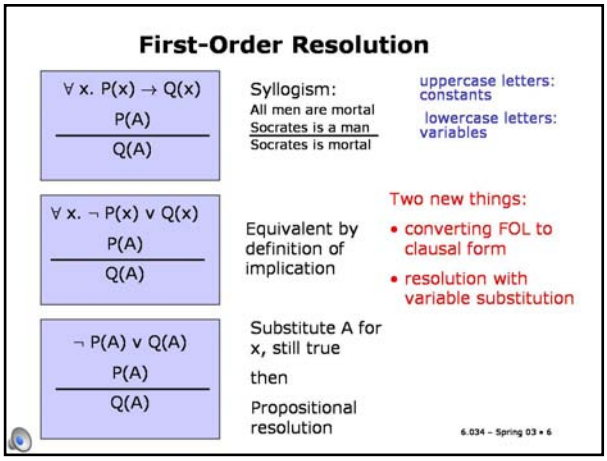


Slide 10.3.4
So, how can we justify this conclusion formally? Well, the first step would be to get rid of the implication.

Slide 10.3.5
Next, we could substitute the constant **A** in for the variable **x** in the universally quantified sentence. By the semantics of universal quantification, that's allowed. A universally quantified statement has to be true of every object in the universe, including whatever object is denoted by the constant symbol **A**. And now, we can apply the propositional resolution rule.

The hard part is figuring out how to instantiate the variables in the universal statements. In this problem, it was clear that **A** was the relevant individual. But it not necessarily clear at all how to do that automatically.





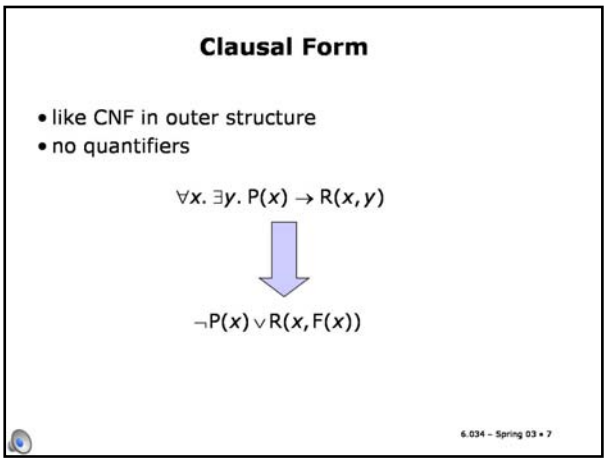
Slide 10.3.6
Now, we have to do two jobs before we can see how to do first-order resolution.

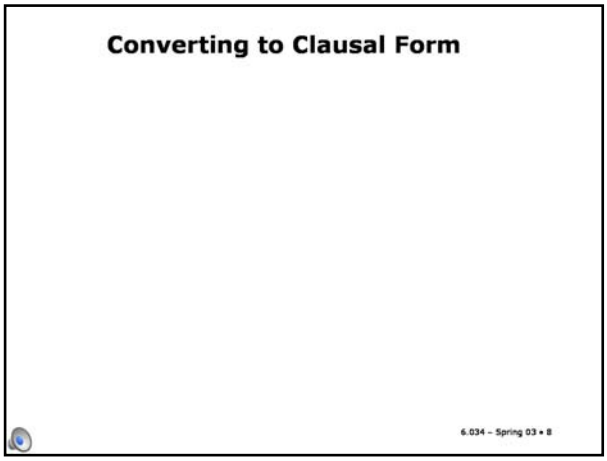
The first is to figure out how to convert from sentences with the whole rich structure of quantifiers into a form that lets us use resolution. We'll need to convert to clausal form, which is a kind of generalization of CNF to first-order logic.

The second is to automatically determine which variables to substitute in for which other ones when we're performing first-order resolution. This process is called unification.

We'll do clausal form next, then unification, and finally put it all together.

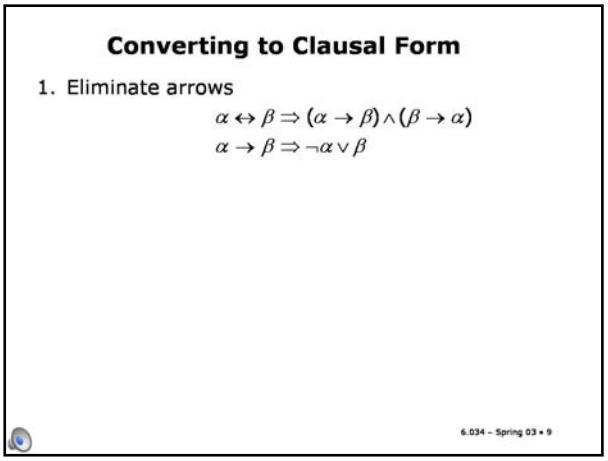
Slide 10.3.7
Clausal form (which is also sometimes called "prenex normal form") is like CNF in its outer structure (a conjunction of disjunctions, or an "and" of "ors"). But it has no quantifiers. Here's an example conversion.

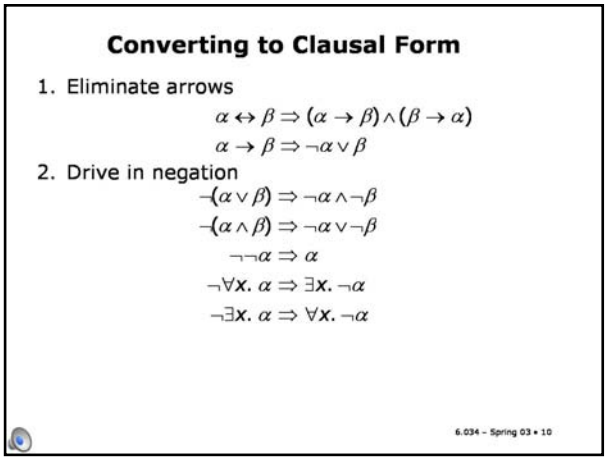




Slide 10.3.8
We'll go through a step-by-step procedure for systematically converting any sentence in first-order logic into clausal form.

Slide 10.3.9
The first step you guys know very well is to eliminate arrows. You already know how to do that. You convert an equivalence into two implications. And anywhere you see alpha right arrow beta, you just change it into not alpha or beta.

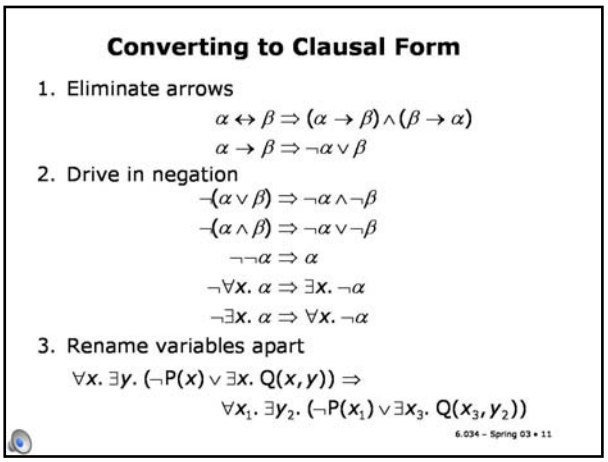


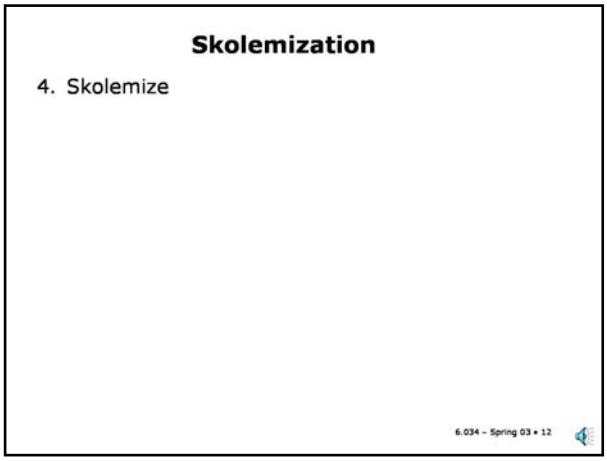


Slide 10.3.10
The next thing you do is drive in negation. You already basically know how to do that. We have deMorgan's laws to deal with conjunction and disjunction, and we can eliminate double negations.

As a kind of extension of deMorgan's laws, we also have that **not (for all x, alpha)** turns into **exists x such that not alpha**. And that **not (exists x such that alpha)** turns into **for all x, not alpha**. The reason these are extensions of deMorgan's laws, in a sense, is that a universal quantifier can be seen abstractly as a conjunction over all possible assignments of **x**, and an existential as a disjunction.

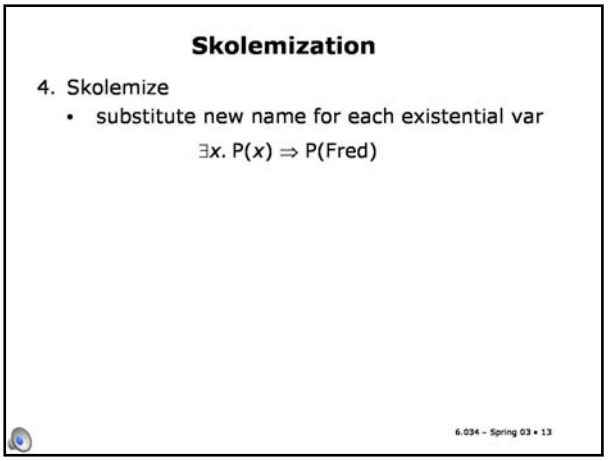
Slide 10.3.11
The next step is to rename variables apart. The idea here is that every quantifier in your sentence should be over a different variable. So, if you had two different quantifications over **x**, you should rename one of them to use a different variable (which doesn't change the semantics at all). In this example, we have two quantifications involving the variable **x**. It's especially confusing in this case, because they're nested. The rules are like those for a programming language: a variable is captured by the nearest enclosing quantifier. So the **x** in **Q(x,y)** is really a different variable from the **x** in **P(x)**. To make this distinction clear, and to automate the downstream processing into clausal form, we'll just rename each of the variables.

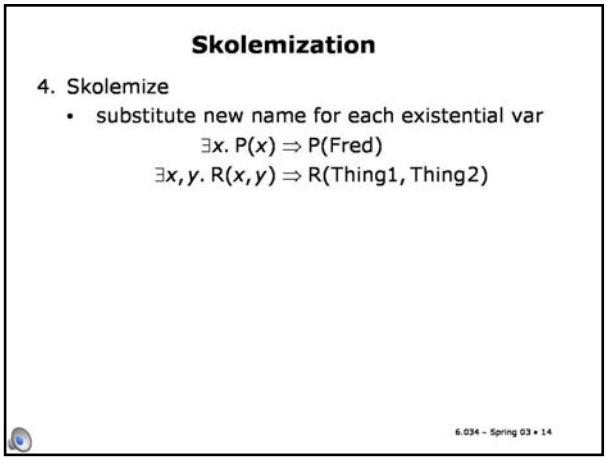




Slide 10.3.12
Now, here's the step that many people find confusing. The name is already a good one. Step four is to skolemize, named after a logician called Thoralf Skolem. Imagine that you have a sentence that looks like: there exists an **x** such that **P(x)**. The goal here is to somehow arrive at a representation that doesn't have any quantifiers in it. Now, if we only had one kind of quantifier in first-order logic, it would be easy because we could just mention variables and all the variables would be implicitly quantified by the kind of quantifier that we have. But because we have two quantifiers, if we dropped all the quantifiers off, there's a mess, because you don't know which kind of quantification is supposed to apply to which variable.

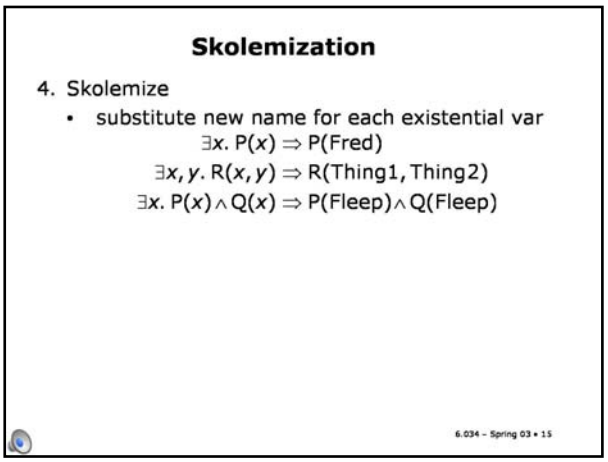
Slide 10.3.13
The Skolem insight is that when you have an existential quantification like this, you're saying there is such a thing as a unicorn, let's say that P means "unicorn". There exists a thing such that it's a unicorn. You can just say, all right, well, if there is one, let's call it Fred. That's it. That's what Skolemization is. So instead of writing exists an **x** such that **P(x)**, you say **P(Fred)**. The trick is that it absolutely must be a new name. It can't be any other name of any other thing that you know about. If you're in the process of inferring things about John and Mary, then it's not good to say, oh, there's a unicorn and it's John -- because that's adding some information to the picture. So to Skolemize, in the simple case, means to substitute a brand-new name for each existentially quantified variable.

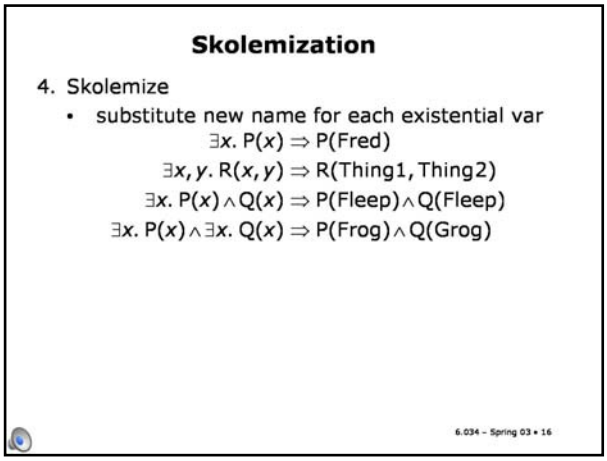




Slide 10.3.14
For example, if I have **exists x, y such that R(x,y)**, then it's going to have to turn into **R(Thing1, Thing2)**. Because we have two different variables here, they have to be given different names.

Slide 10.3.15
But the names also have to persist so that if you have **exists x such that P(x) and Q(x)**, then if you skolemize that expression you should get **P(Fleep)** and **Q(Fleep)**. You make up a name and you put it in there, but every occurrence of this variable has to get mapped into that same unique name.

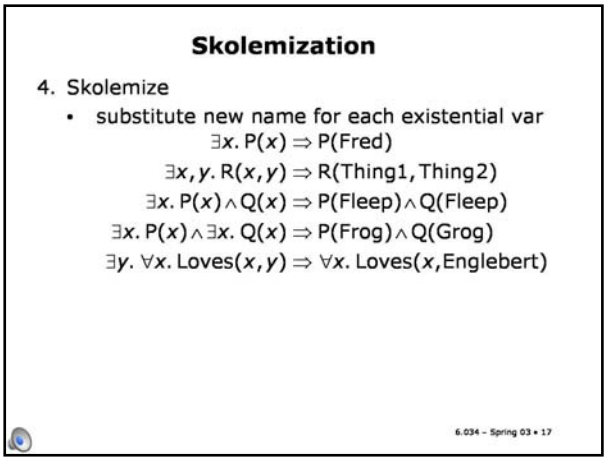


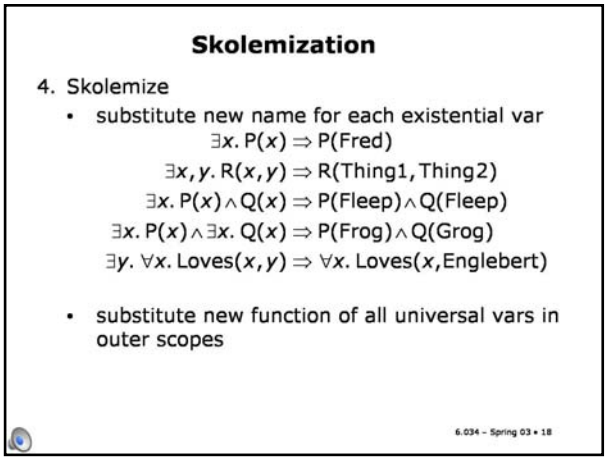


Slide 10.3.16
If you have different quantifiers, then you need to use different names.

Slide 10.3.17
All right. If that's all we had to do it wouldn't be too bad. But there's one more case. We can illustrate it by looking at two interpretations of "Everyone loves someone".

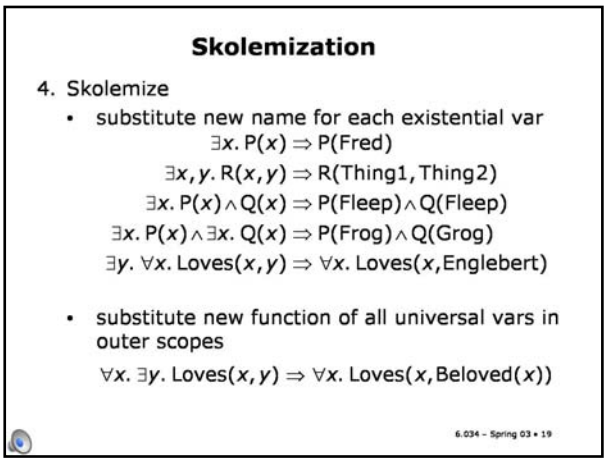
In the first case, there is a single **y** that everyone loves. So we do ordinary skolemization and decide to call that person **Englebert**.

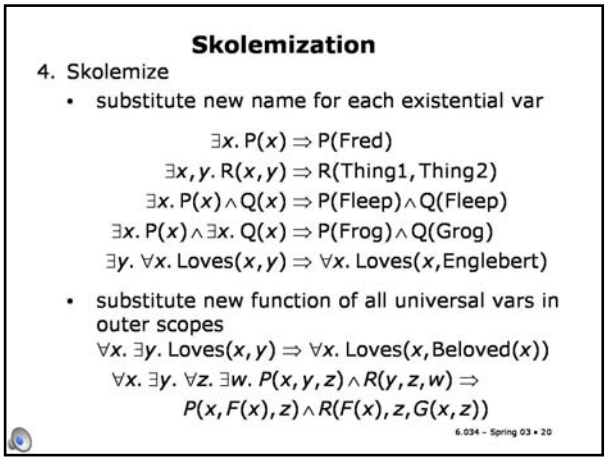




Slide 10.3.18
In the second case, there is a different **y**, potentially, for each **x**. So, if we were just to substitute in a single constant name for **y**, we'd lose that information. We'd get the same result as above, which would be wrong. So, when you are skolemizing an existential variable, you have to look at the other quantifiers that contain the one you're skolemizing, and instead of substituting in a new constant, you substitute in a brand new function symbol, applied to any variables that are universally quantified in an outer scope.

Slide 10.3.19
In this case, what that means is that you substitute in some function of **x**, for **y**. Let's call it **Beloved(x)**. Now it's clear that the person who is loved by **x** depends on the particular **x** you're talking about.

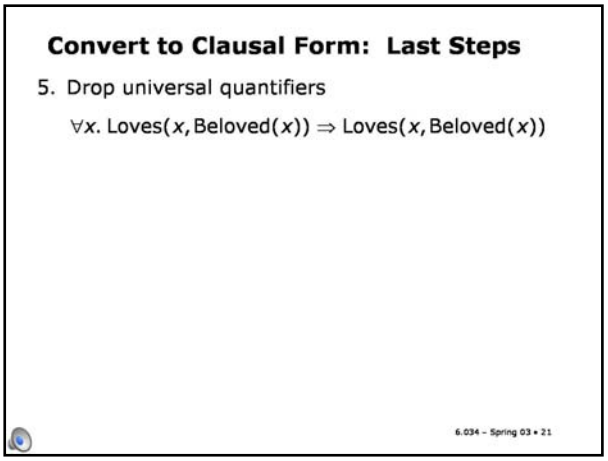


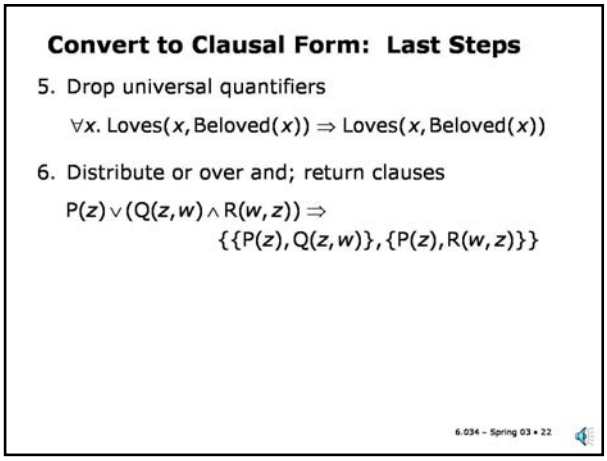


Slide 10.3.20
So, in this example, we see that the existential variable **w** is contained in the scope of two universally quantified variables, **x**, and **z**. So, we replace it with **G(x,z)**, which allows it to depend on the choices of **x** and **z**.

Note also, that I've been using silly names for Skolem constants and functions (like **Englebert** and **Beloved**). But you, or the computer, are only obliged to use new ones, so things like **F123221** are completely appropriate, as well.

Slide 10.3.21
Now we can drop the universal quantifiers because we just replaced all of the existential quantifiers with Skolem constants or functions. Now there's only one kind of quantifier left, so we can just drop them without losing information.

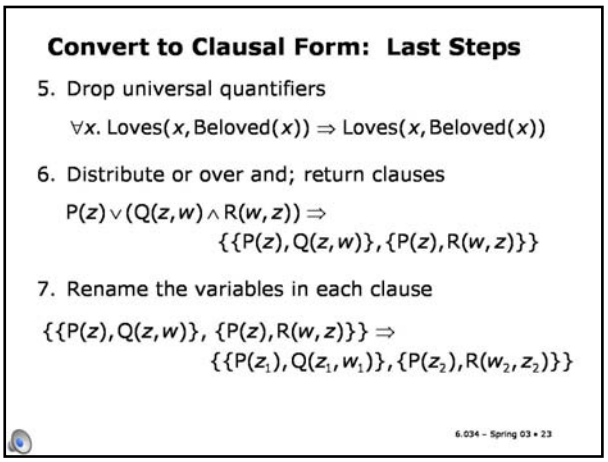


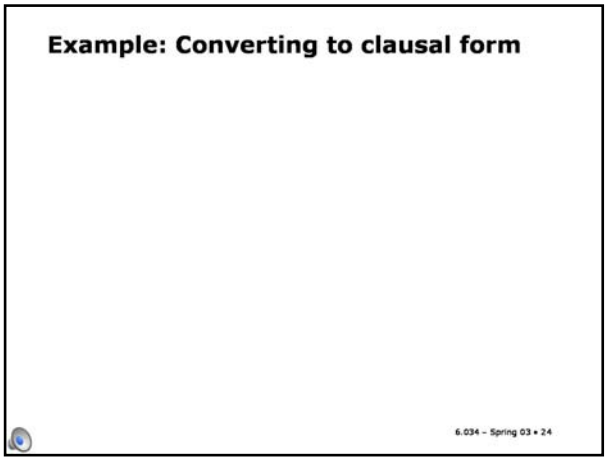


Slide 10.3.22
And then we convert to clauses. This just means multiplying out the and's and the or's, because we already eliminated the arrows and pushed in the negations. We'll return a set of sets of literals. A literal, in this case, is a predicate applied to some terms, or the negation of a predicate applied to some terms.

I'm using set notation here for clauses, just to emphasize that they aren't lists; that the order of the literals within a clause and the order of the clauses within a set of clauses, doesn't have any effect on its meaning.

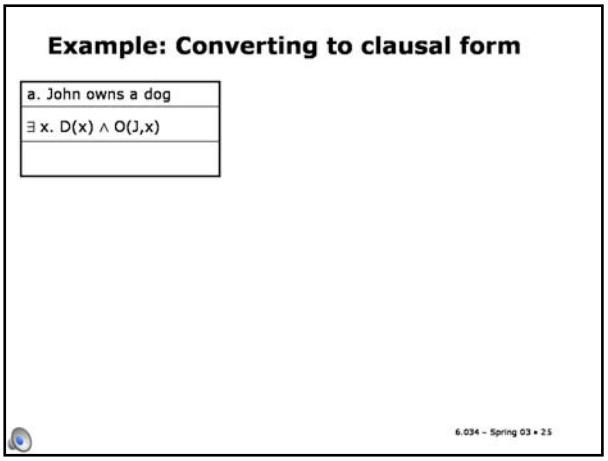
Slide 10.3.23
Finally, we can rename the variables in each clause. It's okay to do that because **for all x, P(x) and Q (x)** is equivalent to **for all y, P(y)** and **for all z, P(z)**. In fact, you don't really need to do this step, because we're assuming that you're always going to rename the variables before you do a resolution step.

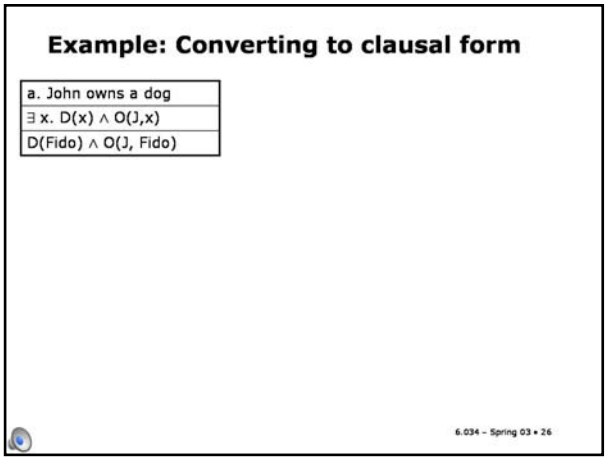




Slide 10.3.24
So, let's do an example, starting with English sentences, writing them down in first-order logic, and converting them to clausal form. Later, we'll do a resolution proof using these clauses.

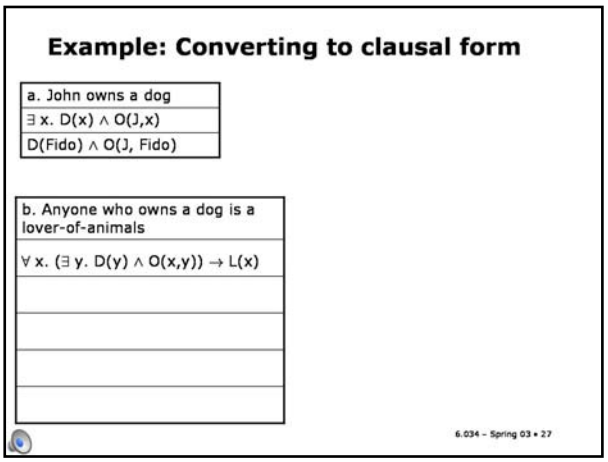
Slide 10.3.25
John owns a dog. We can write that in first-order logic as **there exists an x such that D(x) and O(J, x)**. So, we're letting **D** stand for "is a dog" and **O** stand for "owns" and **J** stand for John.

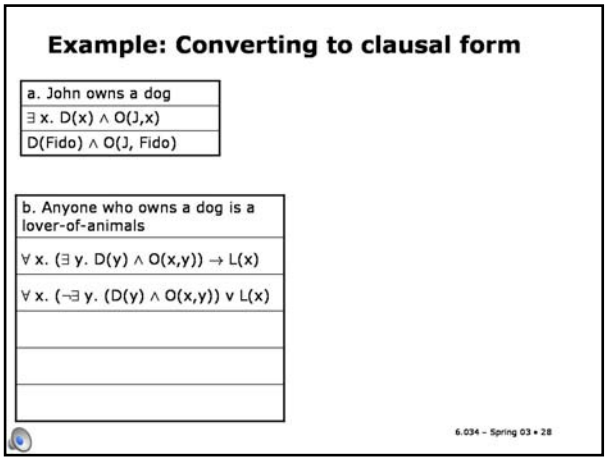




Slide 10.3.26
To convert this to clausal form, we can start at step 4, Skolemization, because the previous three steps are unnecessary for this sentence. Since we just have an existential quantifier over **x**, without any enclosing universal quantifiers, we can simply pick a new name and substitute it in for **x**. Let's call **x Fido**. This will give us two clauses with no variables, and we're done.

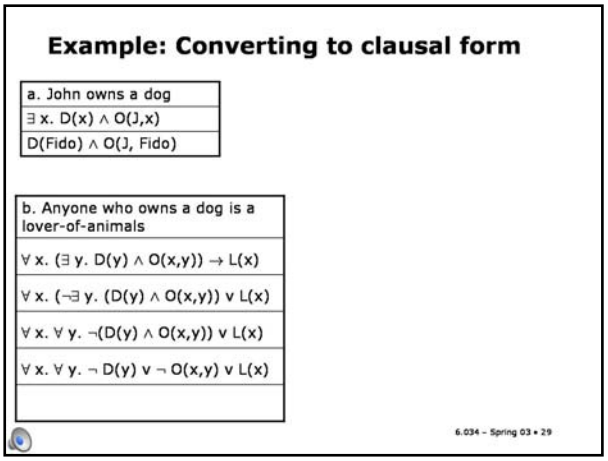
Slide 10.3.27
Anyone who owns a dog is a lover of animals. We can write that in FOL as **For all x, if there exists a y such that D(y) and O(x,y), then L(x)**. We've added a new predicate symbol **L** to stand for "is a lover of animals".

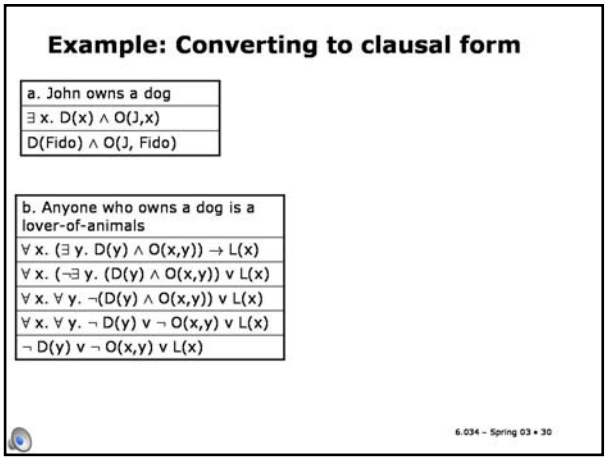




Slide 10.3.28
First, we get rid of the arrow. Note that the parentheses are such that the existential quantifier is part of the antecedent, but the universal quantifier is not. The answer would come out very differently if those parens weren't there; this is a place where it's easy to make mistakes.

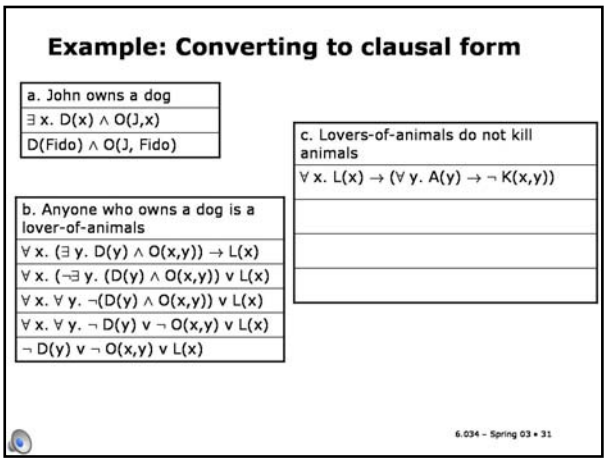
Slide 10.3.29
Next, we drive in the negations. We'll do it in two steps. I find that whenever I try to be clever and skip steps, I do something wrong.

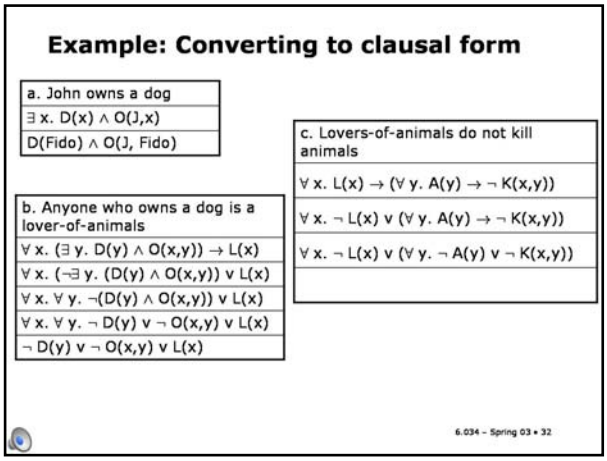




Slide 10.3.30
There's no skolemization to do, since there aren't any existential quantifiers. So, we can just drop the universal quantifiers, and we're left with a single clause.

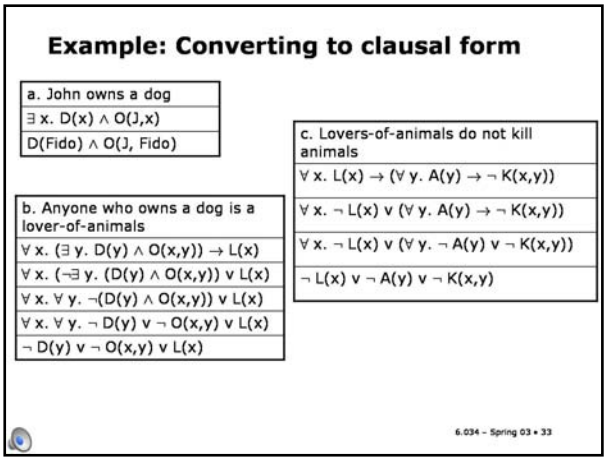
Slide 10.3.31
Lovers of animals do not kill animals. We can write that in FOL as **For all x, L(x) implies that (for all y, A(y) implies not K(x,y))**. We've added the predicate symbol **A** to stand for "is an animal" and the predicate symbol **K** to stand for **x** kills **y**.

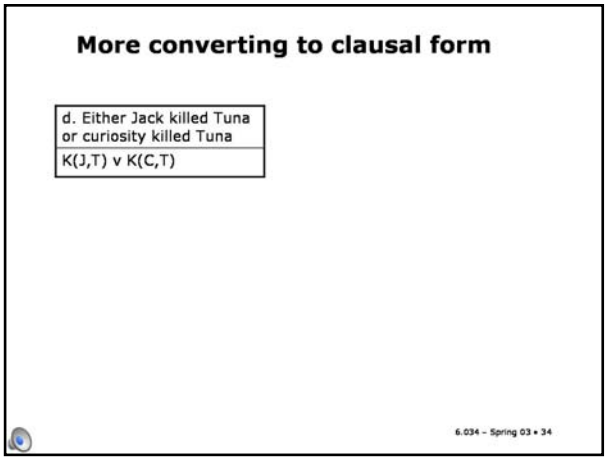




Slide 10.3.32
First, we get rid of the arrows, in two steps.

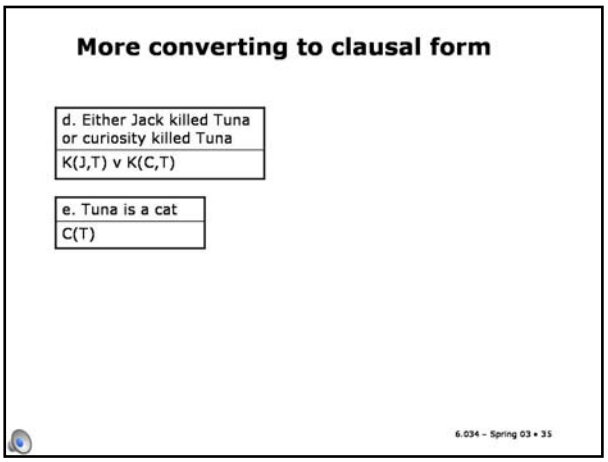
Slide 10.3.33
Then we're left with only universal quantifiers, which we drop, yielding one clause.

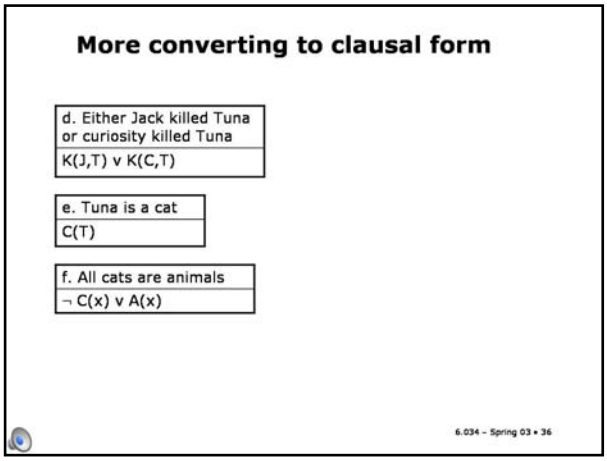




Slide 10.3.34
We just have three more easy ones. "Either Jack killed Tuna or curiosity killed Tuna." Everything here is a constant, so we get **K(J,T) or K(C,T)**.

Slide 10.3.35
"Tuna is a cat" just turns into **C(T)**.



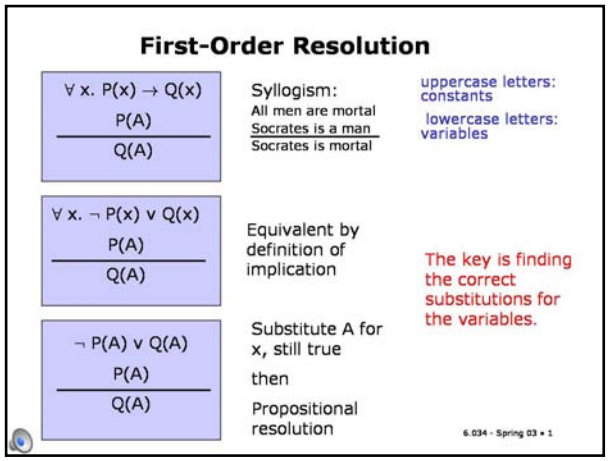


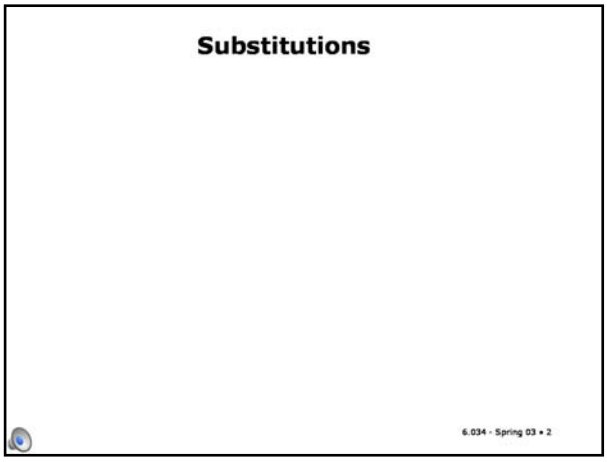
Slide 10.3.36
And "All cats are animals" is **not C(x) or A(x)**. I left out the steps here, but I'm sure you can fill them in.

Okay. Next, we'll see how to match up literals that have variables in them, and move on to resolution.

6.034 Notes: Section 10.4

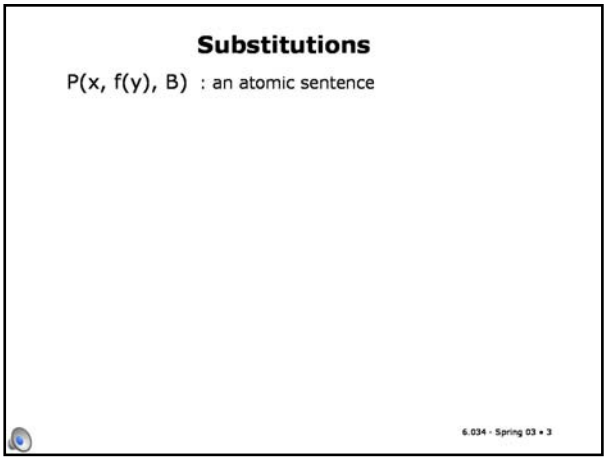
Slide 10.4.1
We introduced first-order resolution and said there were two issues to resolve before we could do it. First was conversion to clausal form, which we've done. Now we have to figure out how to instantiate the variables in the universal statements. In this problem, it was clear that A was the relevant individual. But it is not necessarily clear at all how to do that automatically.

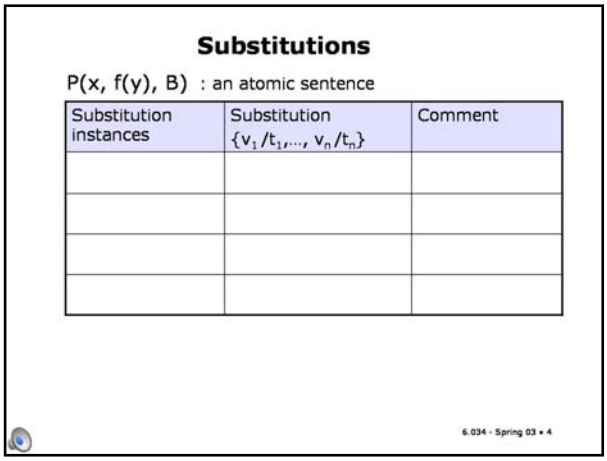




Slide 10.4.2
In order to derive an algorithmic way of finding the right instantiations for the universal variables, we need something called substitutions.

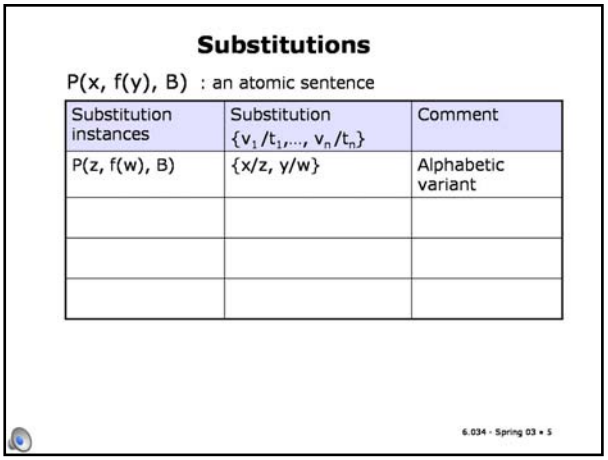
Slide 10.4.3
Here's an example of what we called an atomic sentence before: a predicate applied to some terms. There are two variables here: x and y. We can think of many different ways to substitute terms into this expression. Those are called substitution instances of the expression.

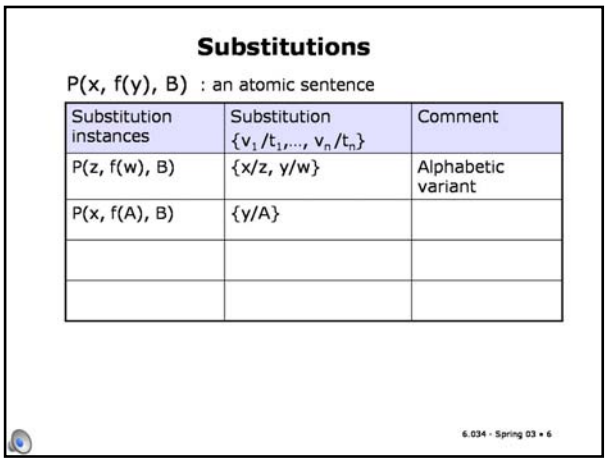




Slide 10.4.4
A substitution is a set of variable-term pairs, written this way. It says that whenever you see variable v_i, you should substitute in term t_i. There should not be more than one entry for a single variable.

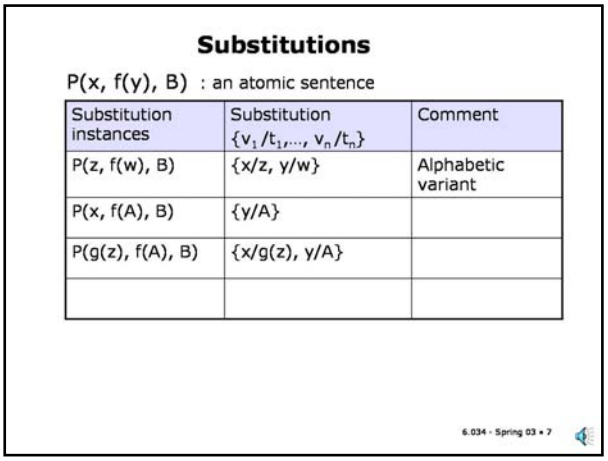
Slide 10.4.5
So here's one substitution instance. P(z,f(w),B). It's not particularly interesting. It's called an alphabetic variant, because we've just substituted some different variables in for x and y. In particular, we've put z in for x and w in for y, as shown in the substitution.

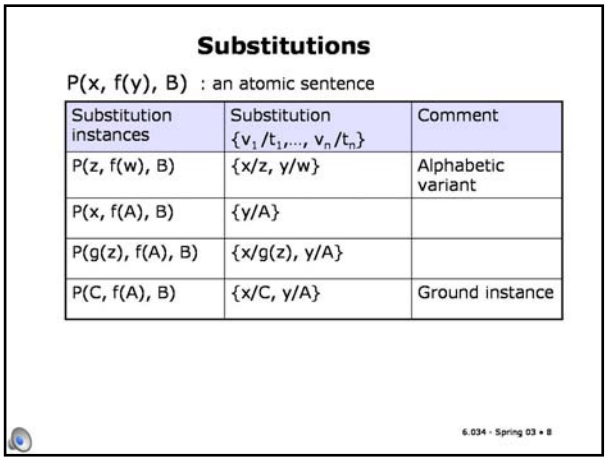




Slide 10.4.6
Here's another substitution instance of our sentence: P(x, f(A), B), We've put the constant A in for the variable y.

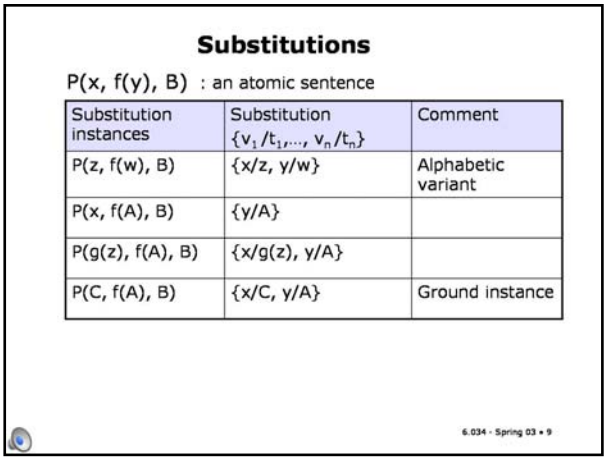
Slide 10.4.7
To get P(g(z), f(A), B), we substitute the term g(z) in for x and the constant A for y.

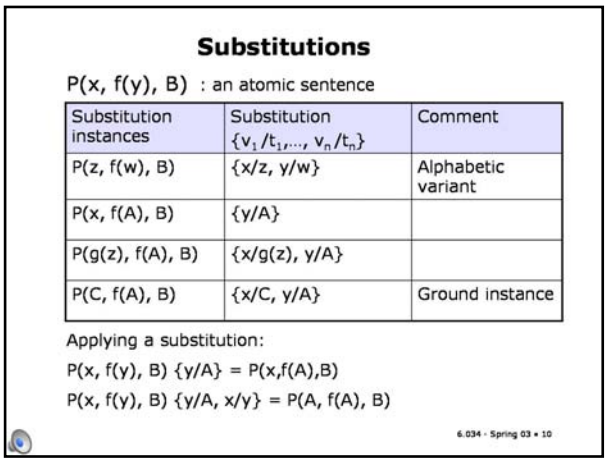




Slide 10.4.8
Here's one more -- P(C, f(A), B). It's sort of interesting, because it doesn't have any variables in it. We'll call an atomic sentence with no variables a ground instance. Ground means it doesn't have any variables.

Slide 10.4.9
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.

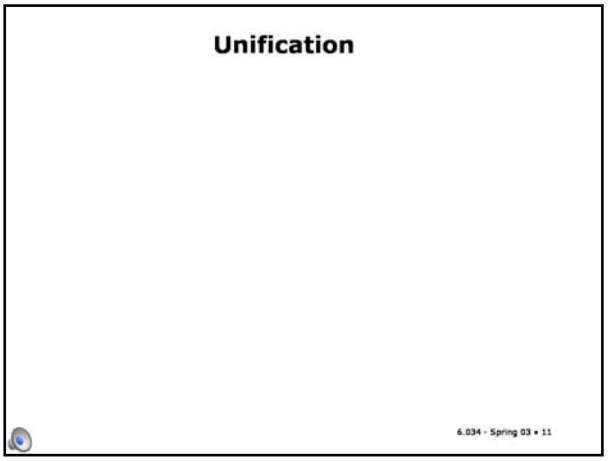


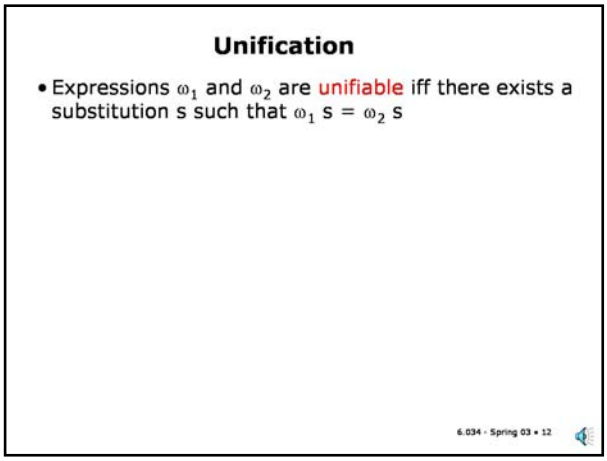


Slide 10.4.10
We'll use the notation of an expression followed by a substitution to mean the expression that we get by applying the substitution to the expression. To apply a substitution to an expression, we look to see if any of the variables in the expression have entries in the substitution. If they 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.

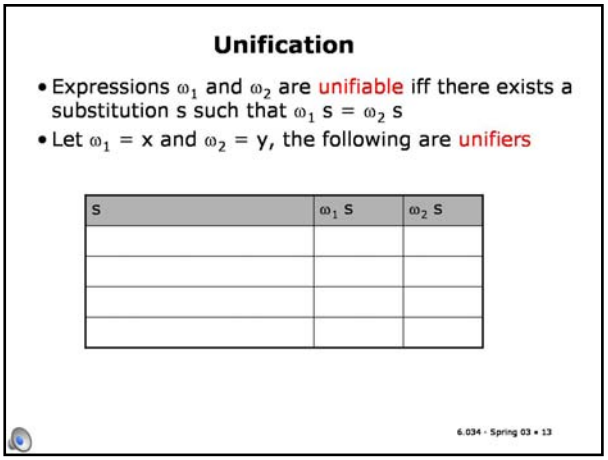
Slide 10.4.11
Now we'll look at the process of unification, which is finding a substitution that makes two expressions match each other exactly.

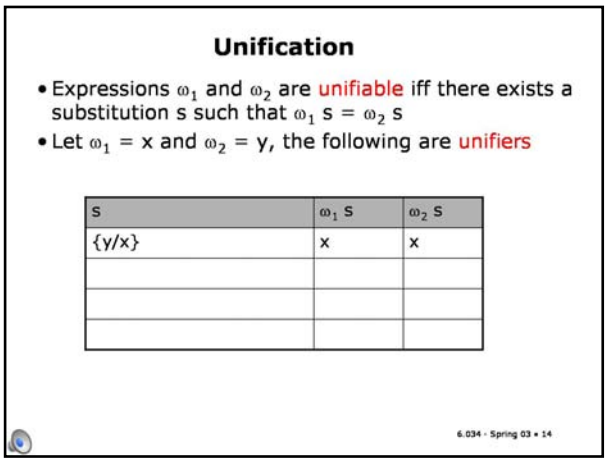




Slide 10.4.12
So, expressions omega$_1$ and omega$_2$ are unifiable if and only if there exists a substitution S such that we get the same thing when we apply S to omega$_1$ as we do when we apply S to omega$_2$. That substitution, S, is called a unifier of omega$_1$ and omega$_2$.

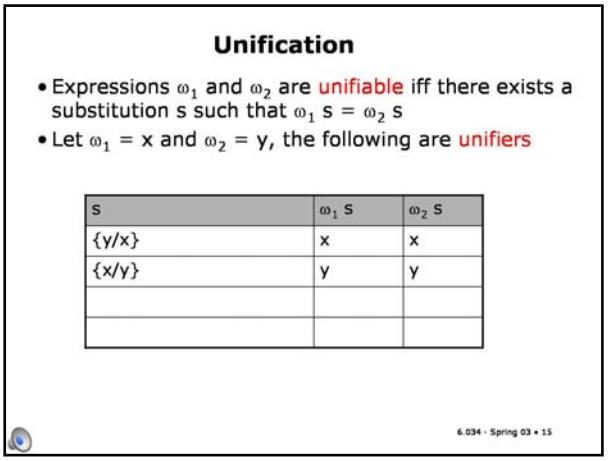
Slide 10.4.13
So, let's look at some unifiers of the expressions x and y. Since x and y are both variables, there are lots of things you can do to make them match.

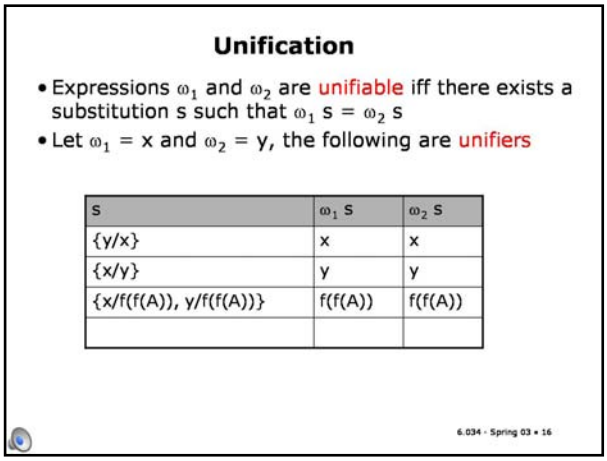




Slide 10.4.14
If you substitute x in for y, then both expressions come out to be x.

Slide 10.4.15
If you put in y for x, then they both come out to be y.

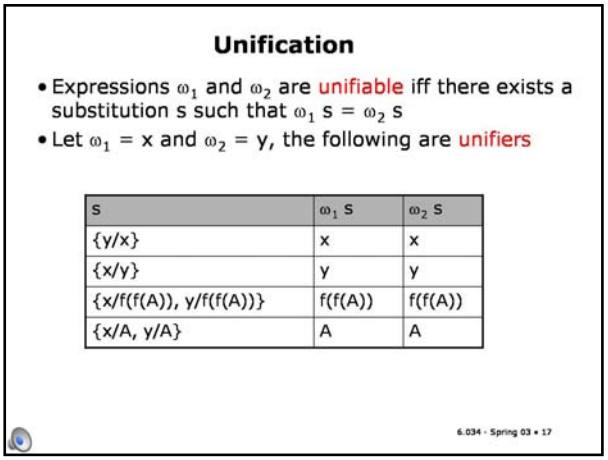


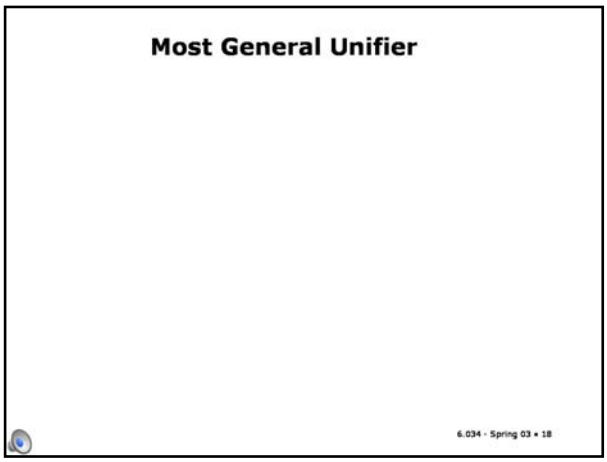


Slide 10.4.16
But you could also substitute something else, like f(f(A)) for x and for y, and you'd get matching expressions.

Slide 10.4.17
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.

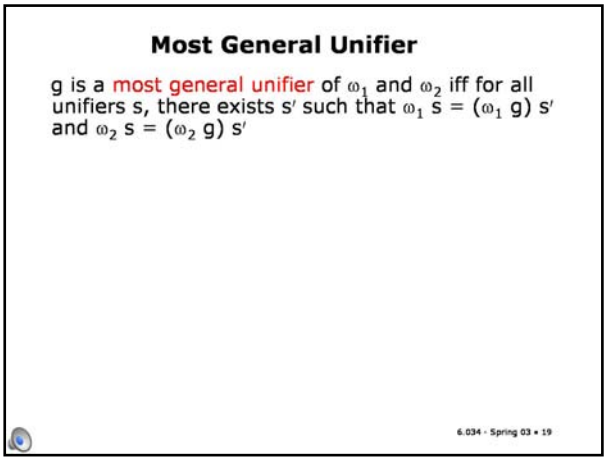


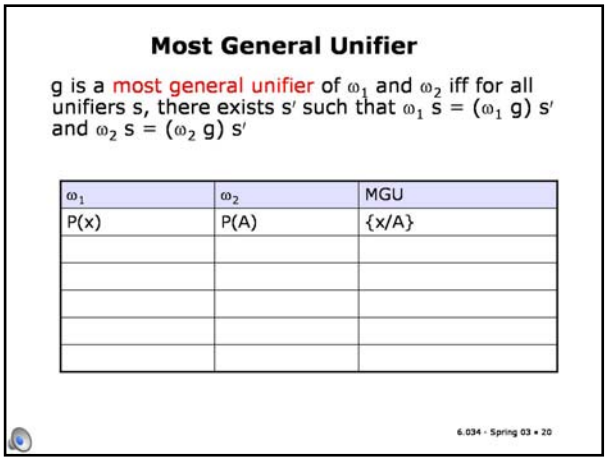


Slide 10.4.18
So, in fact, what we're really going to be looking for is not just any unifier of two expressions, but a most general unifier, or MGU.

Slide 10.4.19
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$_2$.

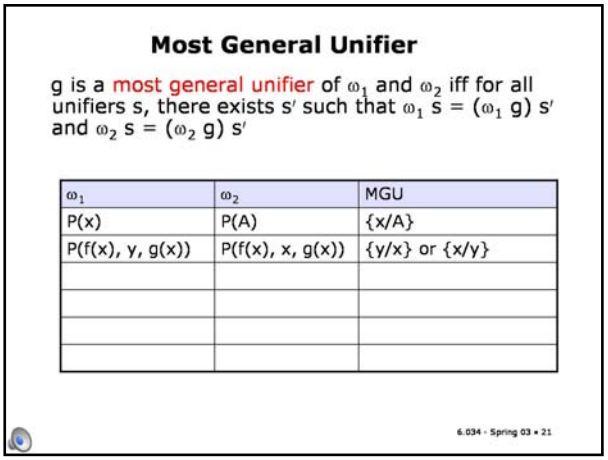
A unifier is most general if every single one of the other unifiers can be expressed as an extra substitution added onto the most general one. An MGU is a substitution that you can make that makes the fewest commitments, and can still make these two expressions equal.

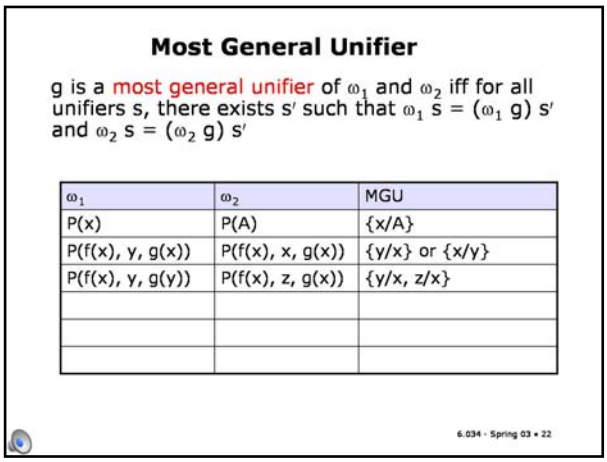




Slide 10.4.20
So, let's do a few examples together. What's a most general unifier of P(x) and P(A)? A for x.

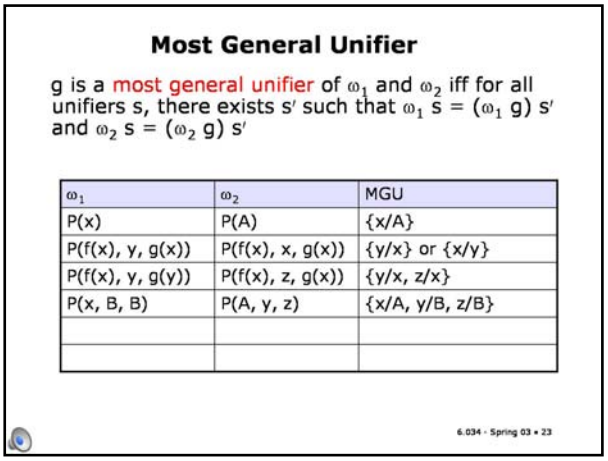
Slide 10.4.21
What about these two expressions? We can make them match up either by substituting x for y, or y for x. It doesn't matter which one we do. They're both "most general".

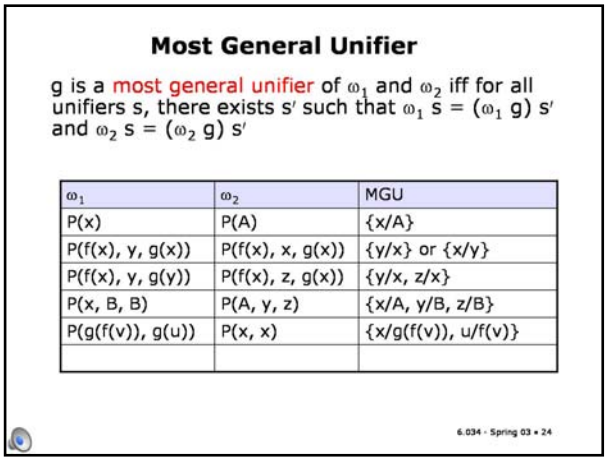




Slide 10.4.22
Okay. What about this one? It's a bit tricky. You can kind of see that, ultimately, all of the variables are going to have to be the same. Matching the arguments to g forces y and x to be the same, And since z and y have to be the same as well (to make the middle argument match), they all have to be the same variable. Might as well make it x (though it could be any other variable).

Slide 10.4.23
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.

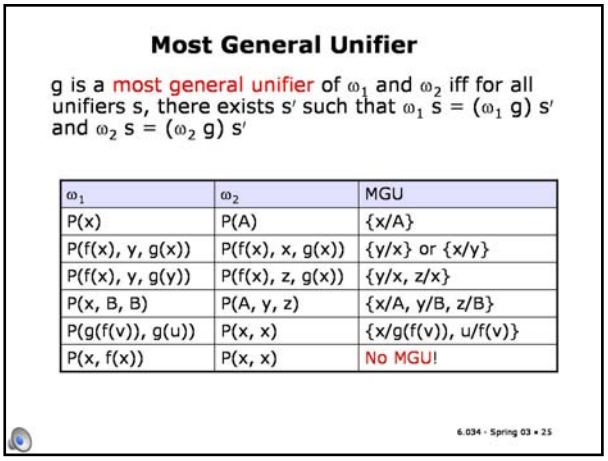


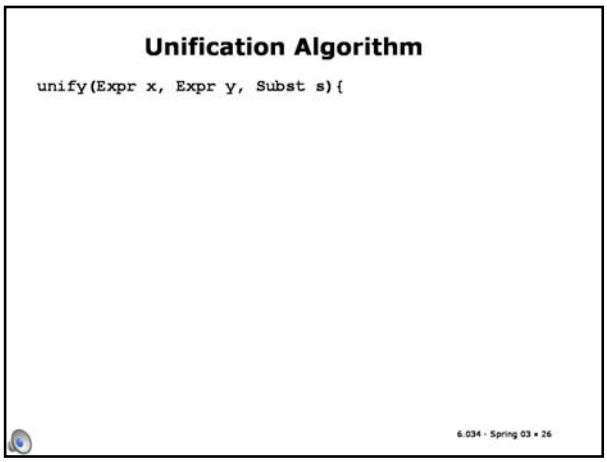


Slide 10.4.24
Here's a tricky one. It looks like x is going to have to simultaneously be g(f(v)) and g(u). How can we make that work? By substituting f(v) in for u.

Slide 10.4.25
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.

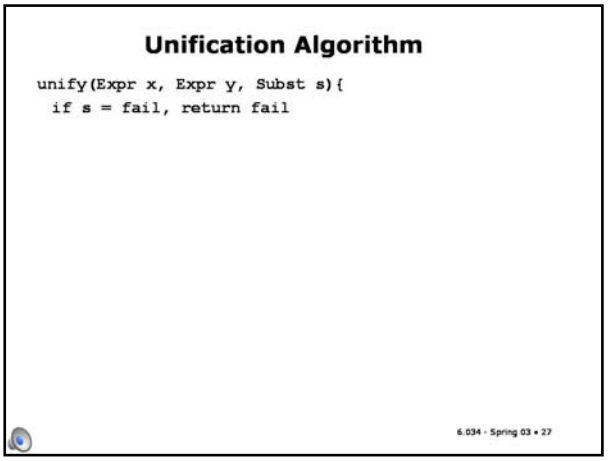
The last time I explained this to a class, someone asked me what would happen if f were the identity function. Then, couldn't we unify these two expressions? That's a great question, and it illustrates a point I should have made before. In unification, we are interested in ways of making expressions 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.

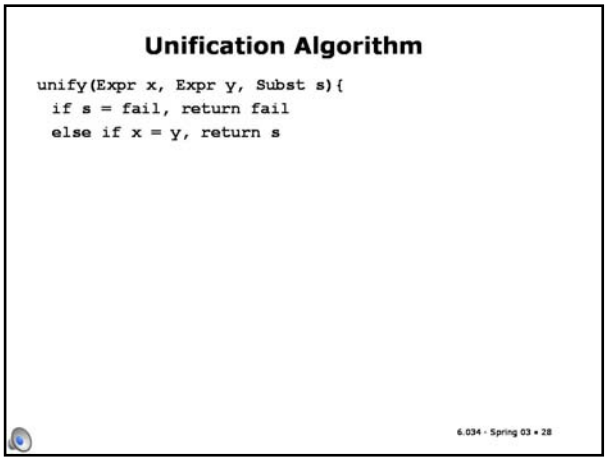




Slide 10.4.26
An MGU can be computed recursively, given two expressions x and y, to be unified, and a substitution that contains substitutions that must already be made. The argument s will be empty in a top-level call to unify two expressions.

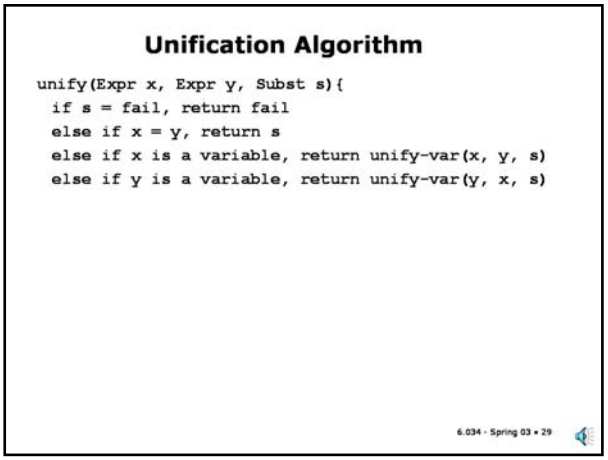
Slide 10.4.27
The algorithm returns a substitution if x and y are unifiable in the context of s, and fail otherwise. If s is already a failure, we return failure.

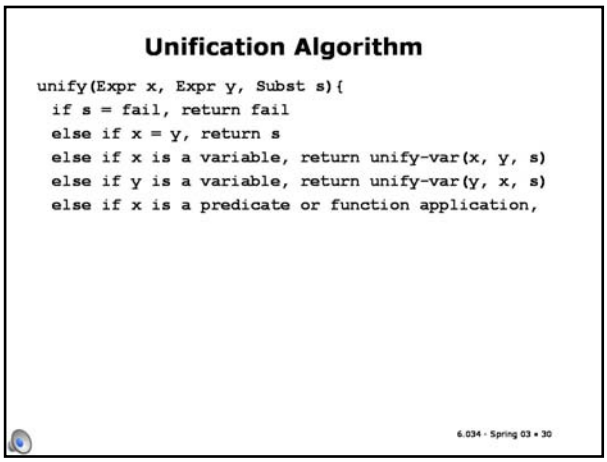




Slide 10.4.28
If x is equal to y, then we don't have to do any work and we return s, the substitution we were given.

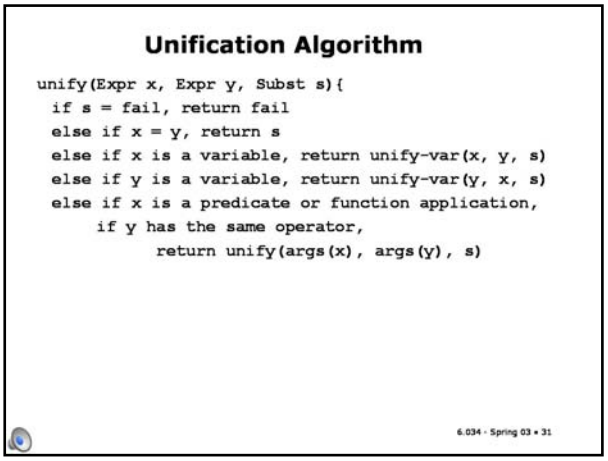
Slide 10.4.29
If either x or y is a variable, then we go to a special subroutine that's shown in upcoming slides.

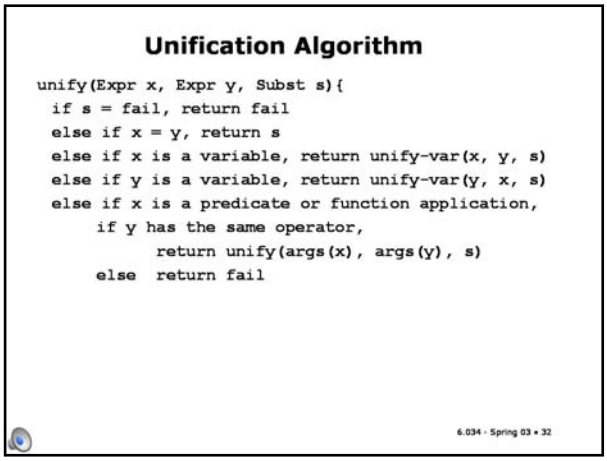




Slide 10.4.30
If x is a predicate or a function application, then y must be one also, with the same predicate or function.

Slide 10.4.31
If so, we'll unify the lists of arguments from x and y in the context of s.

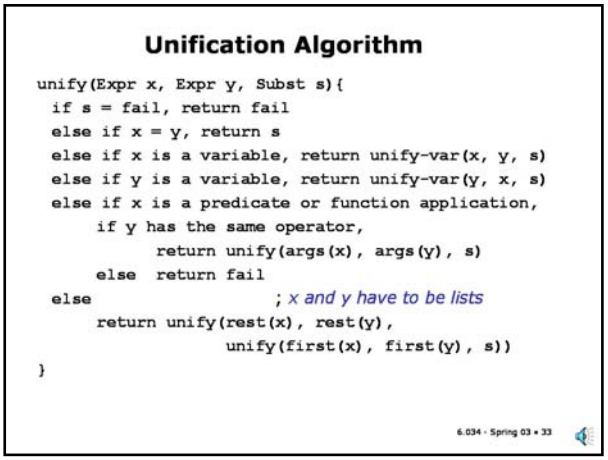


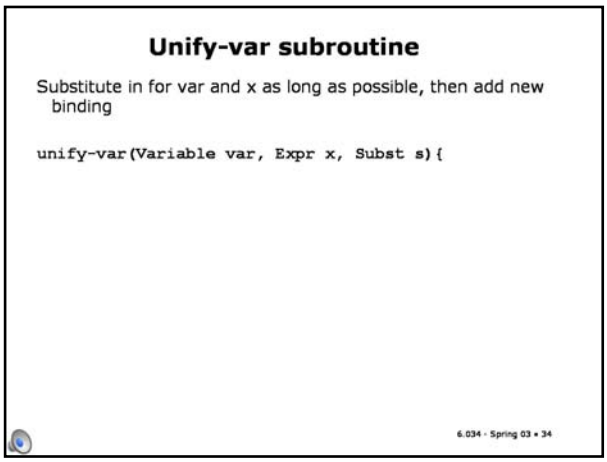


Slide 10.4.32
If not, that is, if x and y have different predicate or function symbols, we simply fail.

Slide 10.4.33
Finally, (if we get to this case, then x and y are either lists of predicate or function arguments, 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.

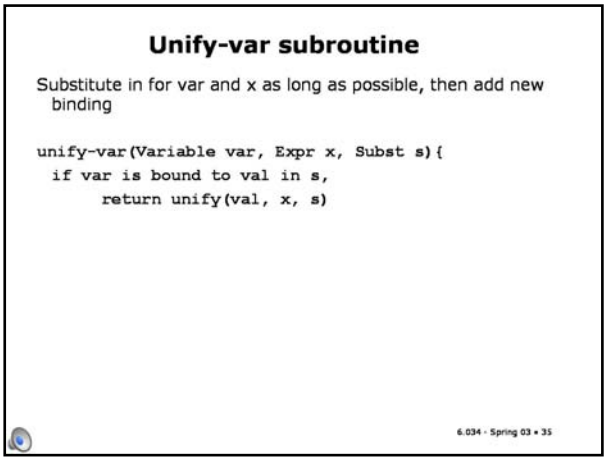
Because, at each stage, we find the most general unifier, we make as few commitments as possible as we go along, and therefore we never have to back up and try a different substitution.

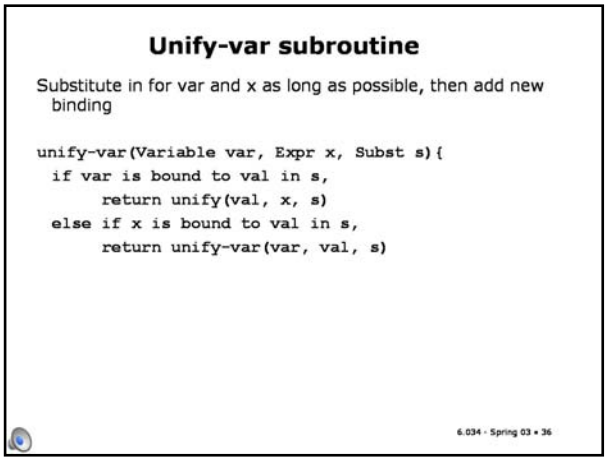




Slide 10.4.34
Given a variable var, an expression x, and a substitution s, we need to return a substitution that unifies var and x in the context of s. What makes this tricky is that we have to first keep applying the existing substitutions in s to var, and to x, if it is a variable, before we're down to a new concrete problem to solve.

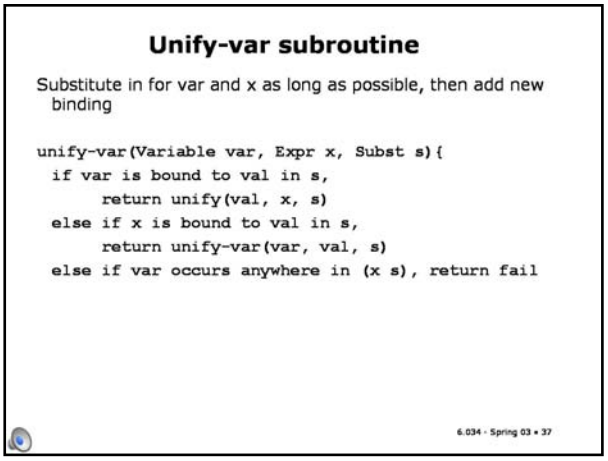
Slide 10.4.35
So, if var is bound to val in s, then we unify that value with x, in the context of s (because we're already committed that val has to be substituted for var).

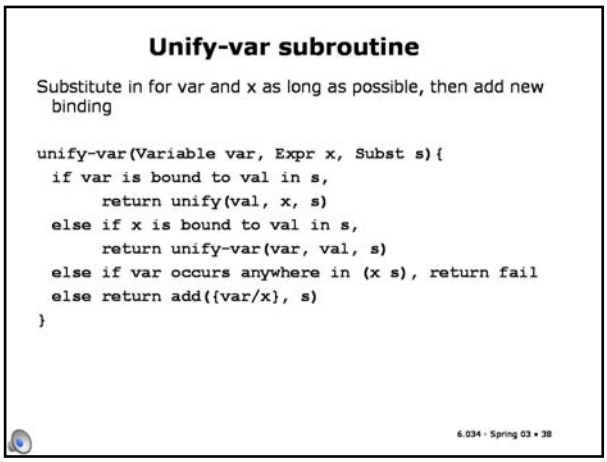




Slide 10.4.36
Similarly, if x is a variable, and it is bound to val in s, then we have to unify var with val in s. (We call unify-var directly, because we know that var is still a var).

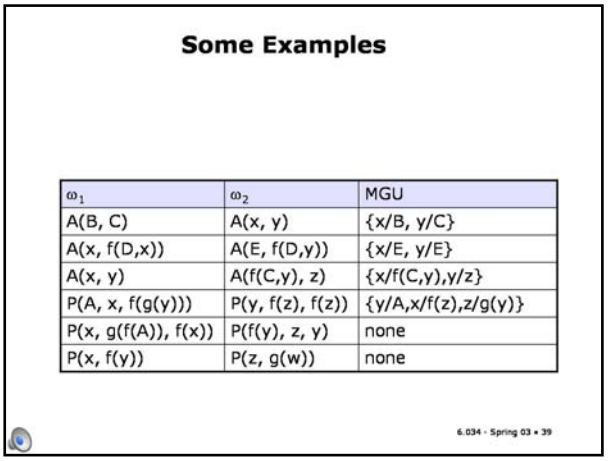
Slide 10.4.37
If var occurs anywhere in x, with substitution s applied to it, then fail. This is the "occurs" check, which keeps us from circularities, like binding x to f(x).





Slide 10.4.38
Finally, we know var is a variable that doesn't already have a substitution, so we add the substitution of x for var to s, and return it.

Slide 10.4.39
Here are a few more examples of unifications, just so you can practice. If you don't see the answer immediately, try simulating the algorithm.



ω_1	ω_2	MGU
A(B, C)	A(x, y)	{x/B, y/C}
A(x, f(D,x))	A(E, f(D,y))	{x/E, y/E}
A(x, y)	A(f(C,y), z)	{x/f(C,y),y/z}
P(A, x, f(g(y)))	P(y, f(z), f(z))	{y/A,x/f(z),z/g(y)}
P(x, g(f(A)), f(x))	P(f(y), z, y)	none
P(x, f(y))	P(z, g(w))	none