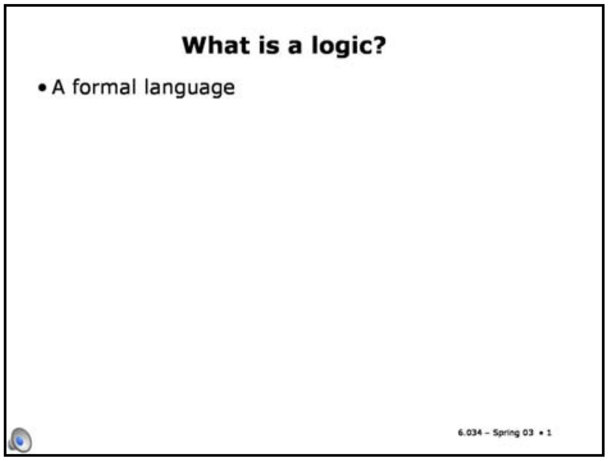
6.034 Notes: Section 9.1

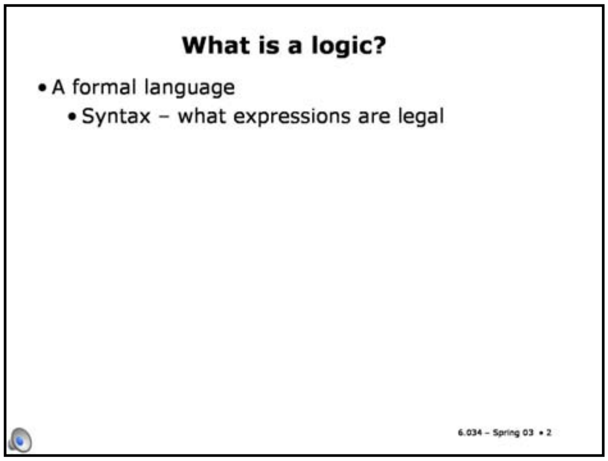
Slide 9.1.1
What is a logic? A logic is a formal language. And what does that mean? It has a syntax and a semantics, and a way of manipulating expressions in the language. We'll talk about each of these in turn.



What is a logic?

- A formal language

6.034 – Spring 03 • 1



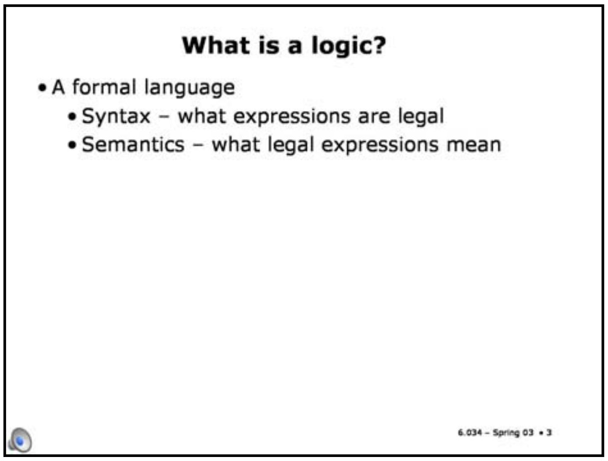
What is a logic?

- A formal language
  - Syntax – what expressions are legal

6.034 – Spring 03 • 2

Slide 9.1.2
The syntax is a description of what you're allowed to write down; what the expressions are that are legal in a language. We'll define the syntax of a propositional logic in complete detail later in this section.

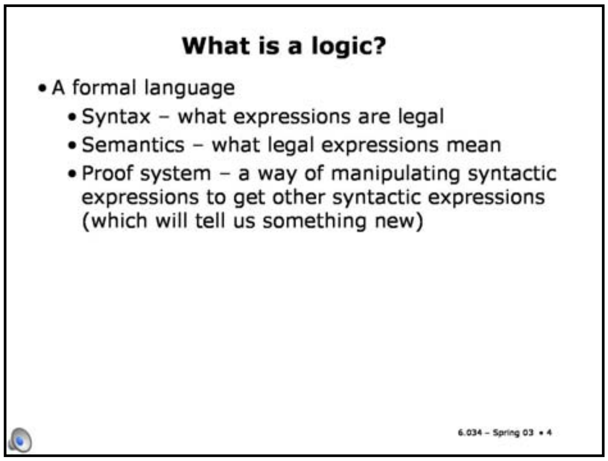
Slide 9.1.3
The semantics is a story about what the syntactic expressions mean. Syntax is form and semantics is content.



What is a logic?

- A formal language
  - Syntax – what expressions are legal
  - Semantics – what legal expressions mean

6.034 – Spring 03 • 3



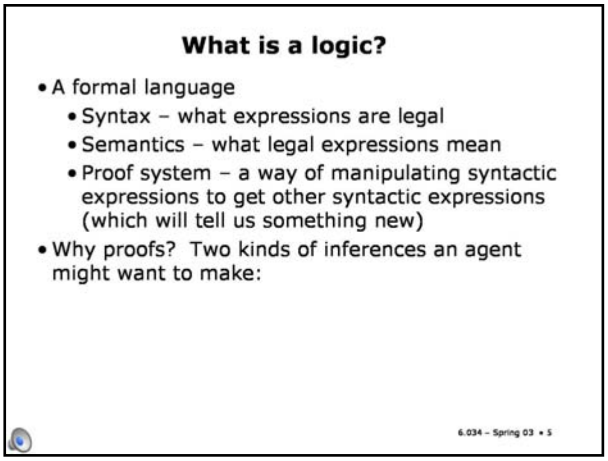
What is a logic?

- A formal language
  - Syntax – what expressions are legal
  - Semantics – what legal expressions mean
  - Proof system – a way of manipulating syntactic expressions to get other syntactic expressions (which will tell us something new)

6.034 – Spring 03 • 4

Slide 9.1.4
A logic usually comes with a proof system, which is a way of manipulating syntactic expressions to get other syntactic expressions. And, why are we interested in manipulating syntactic expressions? The idea is that if we use a proof system with the right kinds of properties, then the new syntactic expressions we create will have semantics or meanings that tell us something "new" about the world.

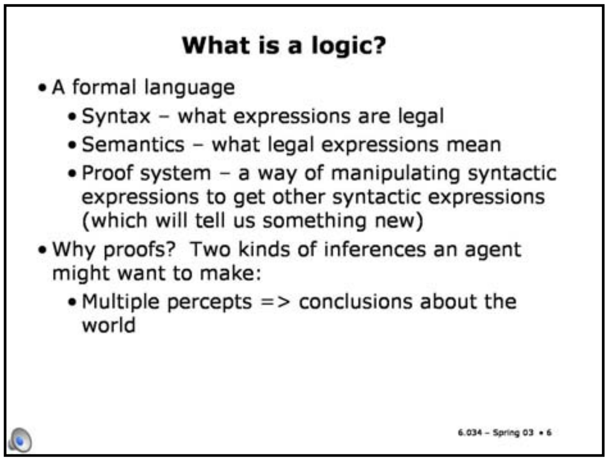
Slide 9.1.5
So, why would we want to do proofs? There are lots of situations.



What is a logic?

- A formal language
  - Syntax – what expressions are legal
  - Semantics – what legal expressions mean
  - Proof system – a way of manipulating syntactic expressions to get other syntactic expressions (which will tell us something new)
- Why proofs? Two kinds of inferences an agent might want to make:

6.034 – Spring 03 • 5



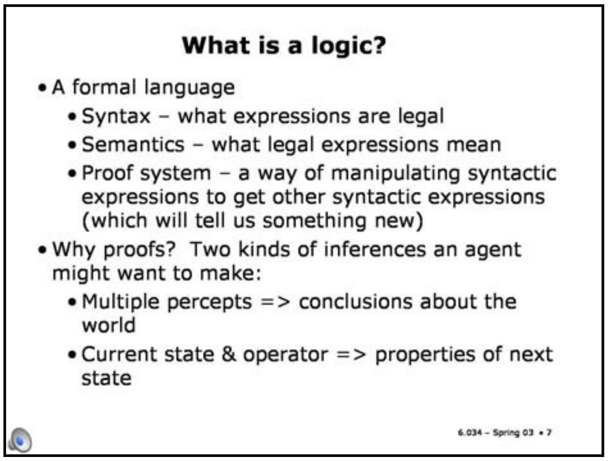
What is a logic?

- A formal language
  - Syntax – what expressions are legal
  - Semantics – what legal expressions mean
  - Proof system – a way of manipulating syntactic expressions to get other syntactic expressions (which will tell us something new)
- Why proofs? Two kinds of inferences an agent might want to make:
  - Multiple percepts => conclusions about the world

6.034 – Spring 03 • 6

Slide 9.1.6
In the context of an agent trying to reason about its world, think about a situation where we have a bunch of percepts. Let's say we saw somebody come in with a dripping umbrella, we saw muddy tracks in the hallway, we see that there's not much light coming in the windows, we hear pitter-pitter-patter. We have all these percepts, and we'd like to draw some conclusion from them, meaning that we'd like to figure out something about what's going on in the world. We'd like to take all these percepts together and draw some conclusion about the world. We could use logic to do that.

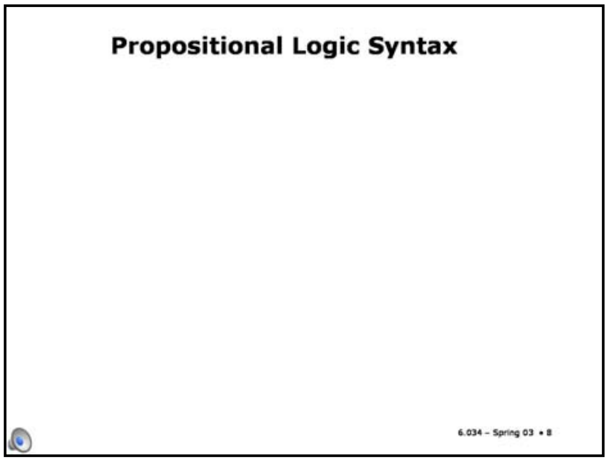
Slide 9.1.7
Another use of logic is when you know something about the current state of the world and you know something about the effects of an action that you're considering doing. You wonder what will happen if you take that action. You have a formal description of what that action does in the world. You might want to take those things together and infer something about the next state of the world. So these are two kinds of inferences that an agent might want to do. We could come up with a lot of other ones, but those are two good examples to keep in mind.



What is a logic?

- A formal language
  - Syntax – what expressions are legal
  - Semantics – what legal expressions mean
  - Proof system – a way of manipulating syntactic expressions to get other syntactic expressions (which will tell us something new)
- Why proofs? Two kinds of inferences an agent might want to make:
  - Multiple percepts => conclusions about the world
  - Current state & operator => properties of next state

6.034 – Spring 03 • 7



Propositional Logic Syntax

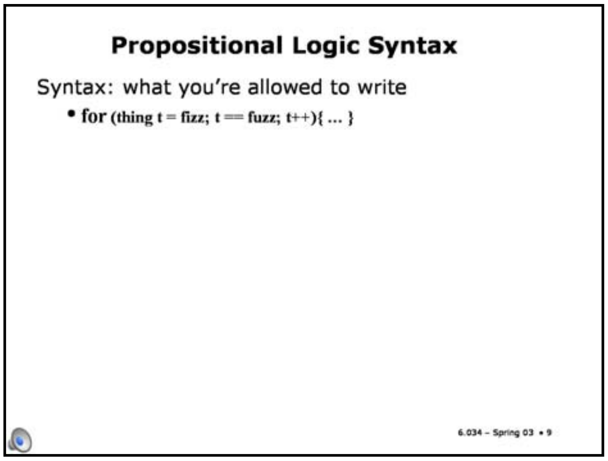
6.034 – Spring 03 • 8

Slide 9.1.8

We'll look at two kinds of logic: propositional logic, which is relatively simple, and first-order logic, which is more complicated. We're just going to dive right into propositional logic, learn something about how that works, and then try to generalize later on. We'll start by talking about the syntax of propositional logic. Syntax is what you're allowed to write on your paper.

Slide 9.1.9

You're all used to rules of syntax from programming languages, right? In Java you can write a for loop. There are rules of syntax given by a formal grammar. They tell you there has to be a semicolon after fizz; that the parentheses have to match, and so on. You can't make random changes to the characters in your program and expect the compiler to be able to interpret it. So, the syntax is what symbols you're allowed to write down in what order. Not what they mean, not what computation they symbolize, but just what symbols you can write down.

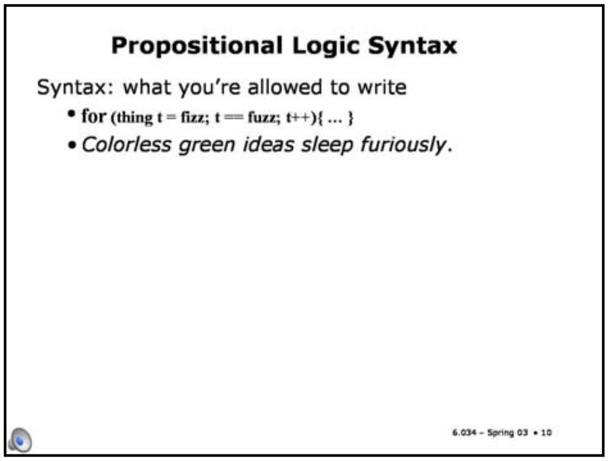


Propositional Logic Syntax

Syntax: what you're allowed to write

- **for** (thing t = fizz; t == fuzz; t++){ … }

6.034 – Spring 03 • 9



Propositional Logic Syntax

Syntax: what you're allowed to write

- **for** (thing t = fizz; t == fuzz; t++){ … }
- *Colorless green ideas sleep furiously.*

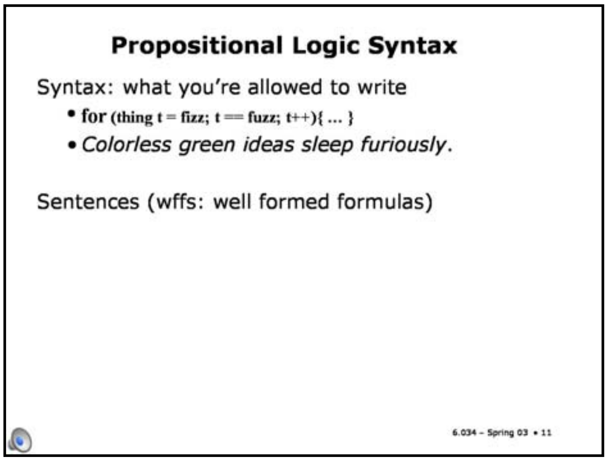
6.034 – Spring 03 • 10

Slide 9.1.10

Another famous illustration of syntax is this one, due to the linguist Noam Chomsky: "Colorless green ideas sleep furiously". The idea is that it doesn't mean anything really, but it's syntactically well-formed. It's got the nouns, the verbs, and the adjectives in the right places. If you scrambled the words up, you wouldn't get a sentence, right? You'd just get a string of words that didn't obey the rules of syntax. So, "furiously ideas green sleep colorless" is not syntactically okay.

Slide 9.1.11

Let's define the syntax of propositional logic. We'll call the legal things to write down "sentences". So if something is a sentence, it is a syntactically okay thing in our language. Sometimes sentences are called "WFFs" (which stands for "well-formed formulas") in other books.



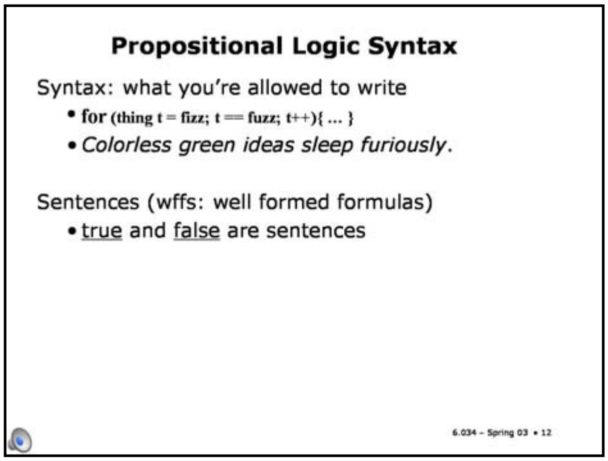
Propositional Logic Syntax

Syntax: what you're allowed to write

- **for** (thing t = fizz; t == fuzz; t++){ … }
- *Colorless green ideas sleep furiously.*

Sentences (wffs: well formed formulas)

6.034 – Spring 03 • 11



Propositional Logic Syntax

Syntax: what you're allowed to write

- for (thing t = fizz; t == fuzz; t++){ … }
- *Colorless green ideas sleep furiously*.

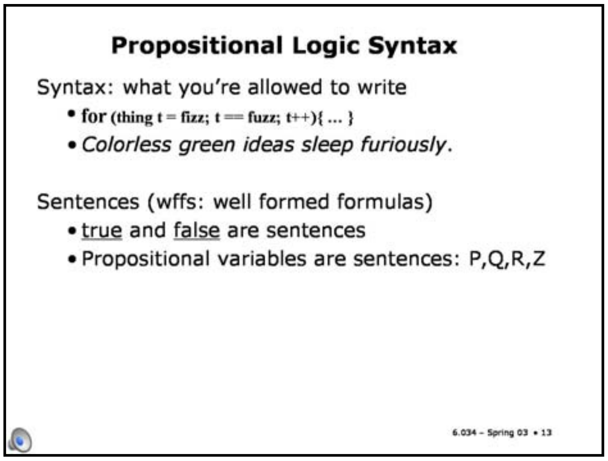
Sentences (wffs: well formed formulas)

- true and false are sentences

6.034 – Spring 03 • 12

Slide 9.1.12
We're going to define the set of legal sentences recursively. So here are two base cases: The words, "true" and "false", are sentences.

Slide 9.1.13
Propositional variables are sentences. I'll give you some examples. P, Q, R, Z. We're not, for right now, defining a language that a computer is going to read. And so we don't have to be absolutely rigorous about what characters are allowed in the name of a variable. But there are going to be things called variables, and we'll just use uppercase letters for them. Those are sentences. It's OK to say "P" -- that's a sentence.



Propositional Logic Syntax

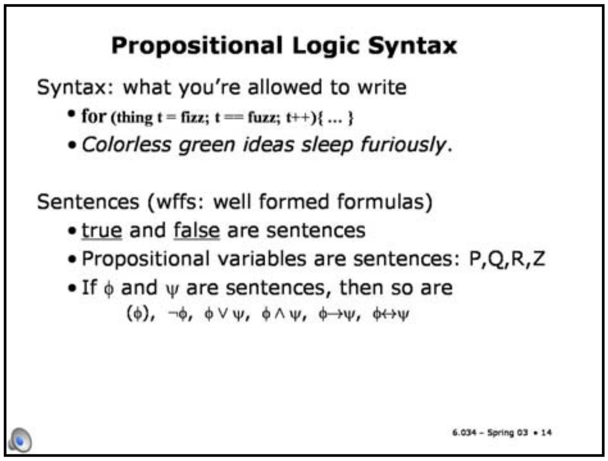
Syntax: what you're allowed to write

- for (thing t = fizz; t == fuzz; t++){ … }
- *Colorless green ideas sleep furiously*.

Sentences (wffs: well formed formulas)

- true and false are sentences
- Propositional variables are sentences: P,Q,R,Z

6.034 – Spring 03 • 13



Propositional Logic Syntax

Syntax: what you're allowed to write

- for (thing t = fizz; t == fuzz; t++){ … }
- *Colorless green ideas sleep furiously*.

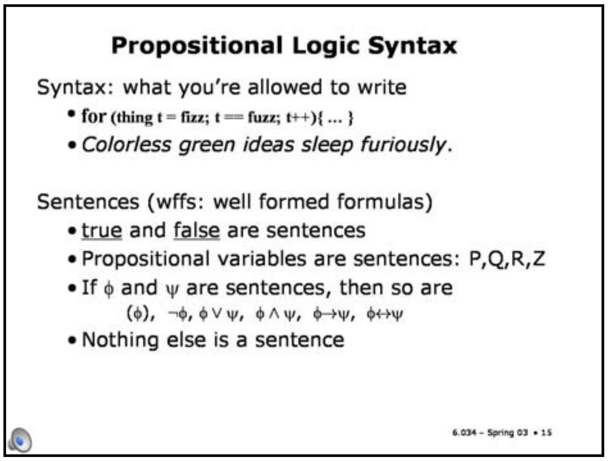
Sentences (wffs: well formed formulas)

- true and false are sentences
- Propositional variables are sentences: P,Q,R,Z
- If ϕ and ψ are sentences, then so are
  (ϕ), $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi$

6.034 – Spring 03 • 14

Slide 9.1.14
Now, here's the recursive part. If Phi and Psi are sentences, then so are -- Wait! What, exactly, are Phi and Psi? They're called metavariables, and they range over expressions. This rule says that if Phi and Psi are things that you already know are sentences because of one of these rules, then you can make more sentences out of them. Phi with parentheses around it is a sentence. Not Phi is a sentence (that little bent thing is our "not" symbol (but we're not really supposed to know that yet, because we're just doing syntax right now)). Phi "vee" Psi is a sentence. Phi "wedge" Psi is a sentence. Phi "arrow" Psi is a sentence. Phi "two-headed arrow" Psi is a sentence.

Slide 9.1.15
And there's one more part of the definition, which says nothing else is a sentence. OK. That's the syntax of the language.



Propositional Logic Syntax

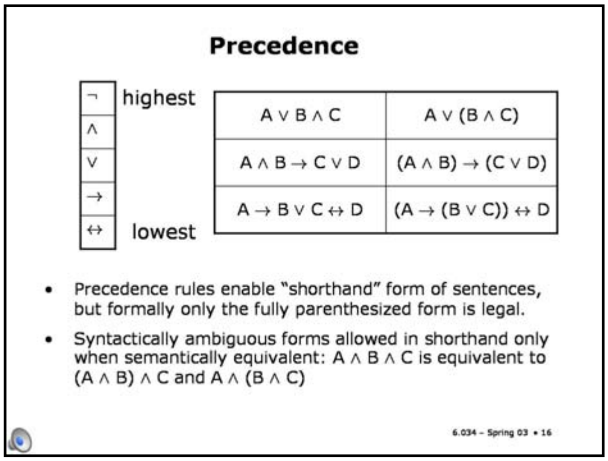
Syntax: what you're allowed to write

- for (thing t = fizz; t == fuzz; t++){ … }
- *Colorless green ideas sleep furiously*.

Sentences (wffs: well formed formulas)

- true and false are sentences
- Propositional variables are sentences: P,Q,R,Z
- If ϕ and ψ are sentences, then so are
  (ϕ), $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi$
- Nothing else is a sentence

6.034 – Spring 03 • 15



Precedence

¬ highest
∧
∨
→
↔ lowest

A ∨ B ∧ C	A ∨ (B ∧ C)
A ∧ B → C ∨ D	(A ∧ B) → (C ∨ D)
A → B ∨ C ↔ D	(A → (B ∨ C)) ↔ D

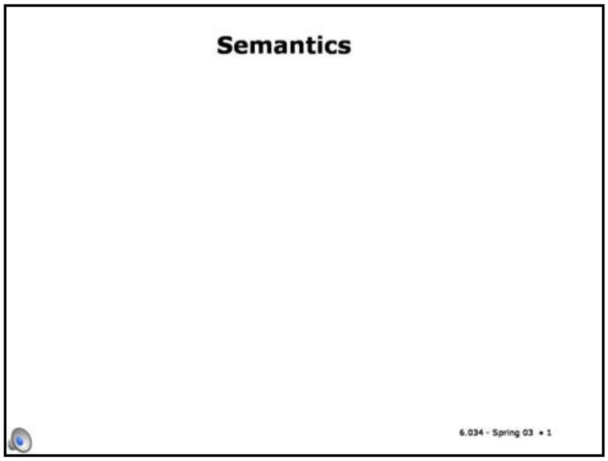
- Precedence rules enable "shorthand" form of sentences, but formally only the fully parenthesized form is legal.
- Syntactically ambiguous forms allowed in shorthand only when semantically equivalent: A ∧ B ∧ C is equivalent to (A ∧ B) ∧ C and A ∧ (B ∧ C)

6.034 - Spring 03 • 16

Slide 9.1.16
There's actually one more issue we have to sort out. Precedence of the operations. If we were being really careful, we'd require you to put parentheses around each new sentence that you made out of component sentences using negation, vee, wedge, or arrow. But it starts getting kind of ugly if we do that. So, we allow you to leave out some of the parentheses, but then we need rules to figure out where the implicit parentheses really are. Those are precedence rules. Just as in arithmetic, where we learned that multiplication binds tighter than addition, we have similar rules in logic. So, to add the parentheses to a sentence, you start with the highest precedence operator, which is negation. For every negation, you'd add an open paren in front of the negation sign and a close parenthesis after the next whole expression. This is exactly how minus behaves in arithmetic. The next highest operator is wedge, which behaves like multiplication in arithmetic. Next is vee, which behaves like addition in arithmetic. Logic has two more operators, with weaker precedence. Next comes single arrow, and last is double arrow. Also, wedge and vee are associative.

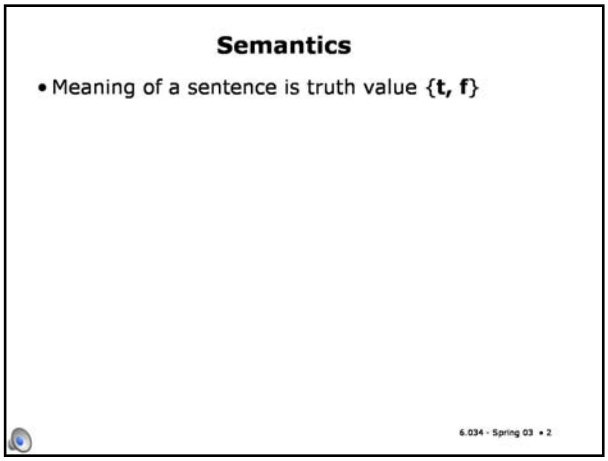
6.034 Notes: Section 9.2

Slide 9.2.1
Let's talk about semantics. The semantics of a sentence is its meaning. What does it say about the world? We could just write symbols on the board and play with them all day long, and it could be fun; it could be like doing puzzles. But ultimately the reason that we want to be doing something with these kinds of logical sentences is because they somehow say something about the world. And it's really important to be clear about the connections between the things that we write on the board and what we think of them as meaning in the world, what they stand for. And it's going to be something different every day. I remember once when I was a little kid, I was on the school bus. And somebody's big sister or brother had started taking algebra and this kid told me, "You know what? My big sister's taking algebra and A equals 3!" The reason that sounds so silly is that A is a variable. Our variables are going to be the same. They'll have different interpretations in different situations. So, in our study of logic, we're not going to assign particular values or meanings to the variables; rather, we're going to study the general properties of symbols and their potential meanings.



Semantics

6.034 - Spring 03 • 1



Semantics

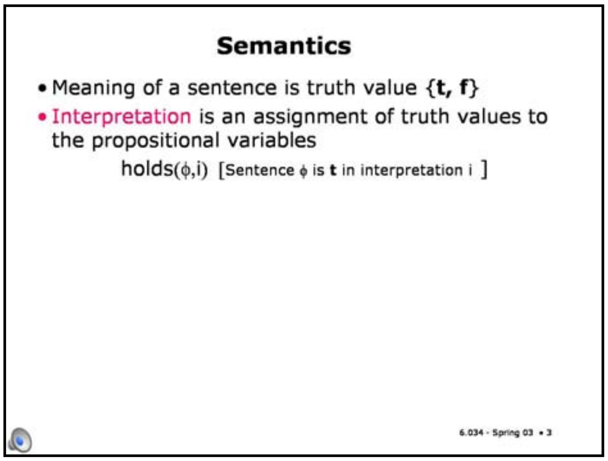
- Meaning of a sentence is truth value {**t**, **f**}

6.034 - Spring 03 • 2

Slide 9.2.2
Ultimately, the meaning of every sentence, in a situation, will be a truth value, **t** or **f**. Just as, in high-school algebra, the meaning of every expression is a numeric value. Note that there's already a really important difference between underlined true and false, which are syntactic entities that we can write on the board, and the truth values **t** and **f** which stand for the abstract philosophical ideals of truth and falsity.

Slide 9.2.3

How can we decide whether A "wedge" B "wedge" C is true or not? Well, it has to do with what A and B and C stand for in the world. What A and B and C stand for in the world will be given by an object called an "interpretation". An interpretation is an assignment of truth values to the propositional variables. You can think of it as a possible way the world could be. So if our set of variables is P, Q, R, and V, then P true, Q false, R true, V true, that would be an interpretation. So then, given an interpretation, we can ask the question, is this sentence true in that interpretation? We will write "holds Phi comma i" to mean "sentence Phi is true in interpretation i". The "holds" symbol is not part of our language. It's part of the way logicians write things on the board when they're talking about what they're doing. This is a really important distinction. If you can think of our sentences like expressions in a programming language, then you can think of these expressions with "holds" as being about whether programs work in a certain way or not. In order to even think about whether Phi is true in interpretation I, Phi has to be a sentence. If it's not a well-formed sentence, then it doesn't even make sense to ask whether it's true or false.

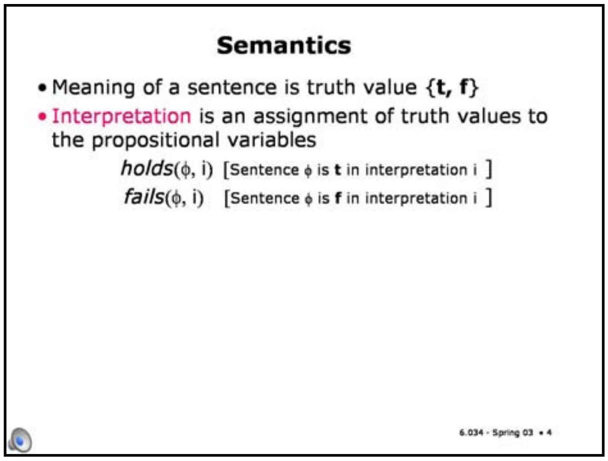


Semantics

- Meaning of a sentence is truth value {**t, f**}
- Interpretation is an assignment of truth values to the propositional variables

  holds(ϕ,i) [Sentence ϕ is **t** in interpretation i]

6.034 - Spring 03 • 3



Semantics

- Meaning of a sentence is truth value {**t, f**}
- Interpretation is an assignment of truth values to the propositional variables

  holds(ϕ, i) [Sentence ϕ is **t** in interpretation i]

  fails(ϕ, i) [Sentence ϕ is **f** in interpretation i]

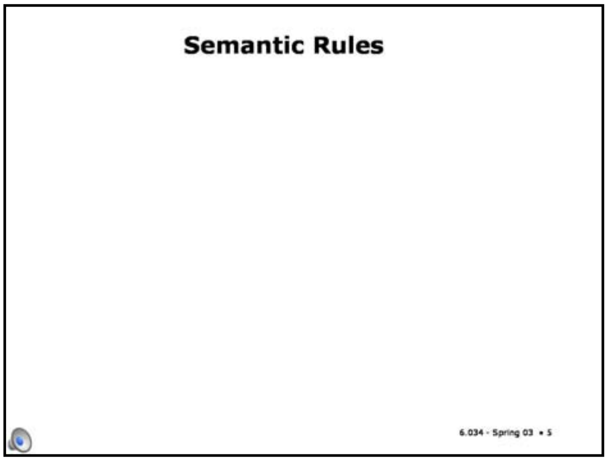
6.034 - Spring 03 • 4

Slide 9.2.4

Similarly, we'll use "fails" to say that a sentence is not true in an interpretation. And since the meaning of every sentence is a truth value and there are only two truth values, then if a sentence Phi is not true (does not have the truth value **t**) in an interpretation, then it has truth value **f** in that interpretation and we'll say it's false in that interpretation.

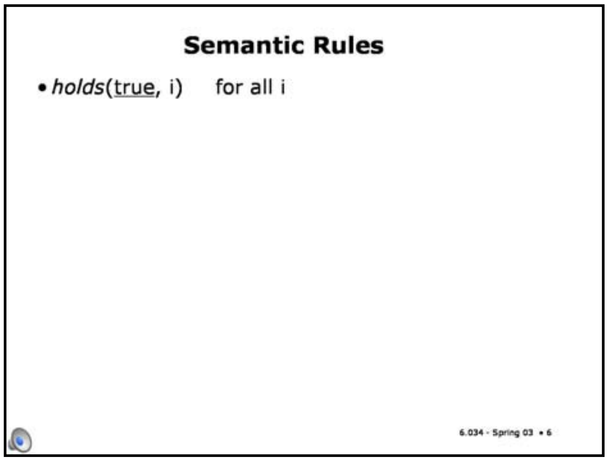
Slide 9.2.5

So now we can write down the rules of the semantics. We can write down rules that specify when sentence Phi is true in interpretation i. We are going to specify the semantics of sentences recursively, based on their syntax. The definition of a semantics should look familiar to most of you, since it's very much like the specification of an evaluator for a functional programming language, such as Scheme.



Semantic Rules

6.034 - Spring 03 • 5



Semantic Rules

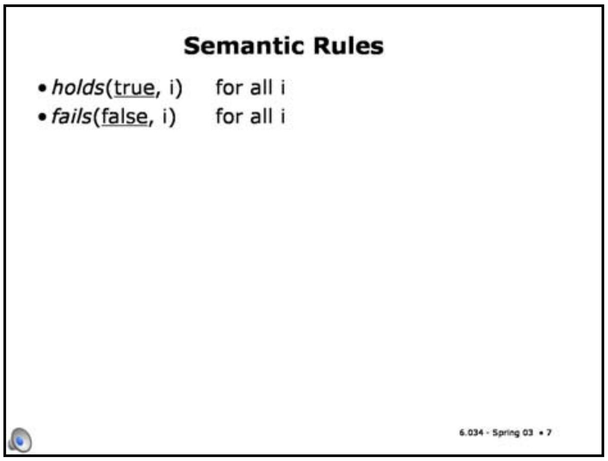
- *holds*(true, i) for all i

6.034 - Spring 03 • 6

Slide 9.2.6

First, the sentence consisting of the symbol "true" is true in all interpretations.

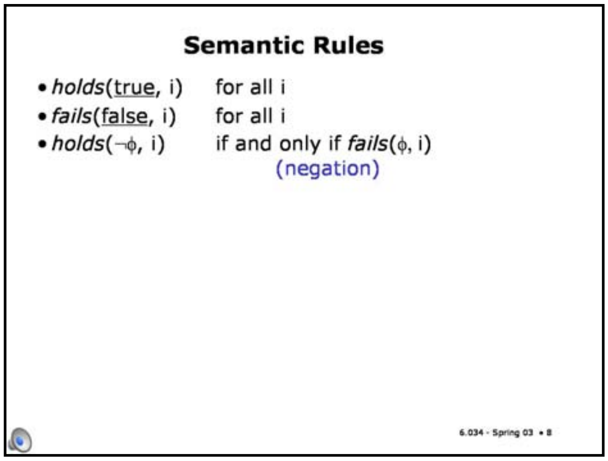
Slide 9.2.7
The sentence consisting of a symbol "false" has truth value **f** in all interpretations.



Semantic Rules

- *holds*(<u>true</u>, i) for all i
- *fails*(<u>false</u>, i) for all i

6.034 - Spring 03 • 7



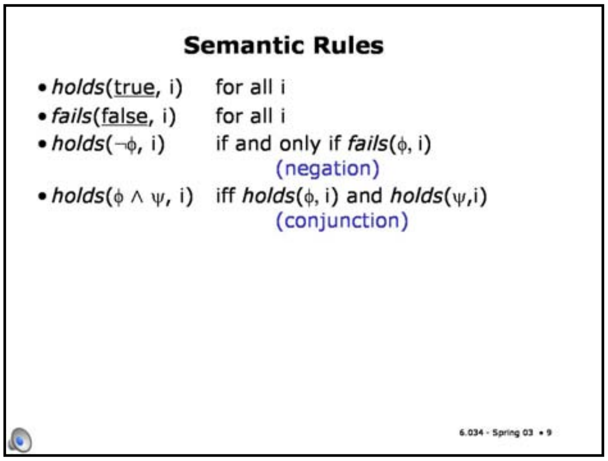
Semantic Rules

- *holds*(<u>true</u>, i) for all i
- *fails*(<u>false</u>, i) for all i
- *holds*($\neg\phi$, i) if and only if *fails*(ϕ, i)
  (negation)

6.034 - Spring 03 • 8

Slide 9.2.8
Now we can do the connectives. We'll leave out the parentheses. The truth value of a sentence with top-level parentheses is the same as the truth value of the sentence with the parentheses removed. Now, let's think about the "not" sign. When is "not" Phi true in an interpretation i? Whenever Phi is false in that interpretation.

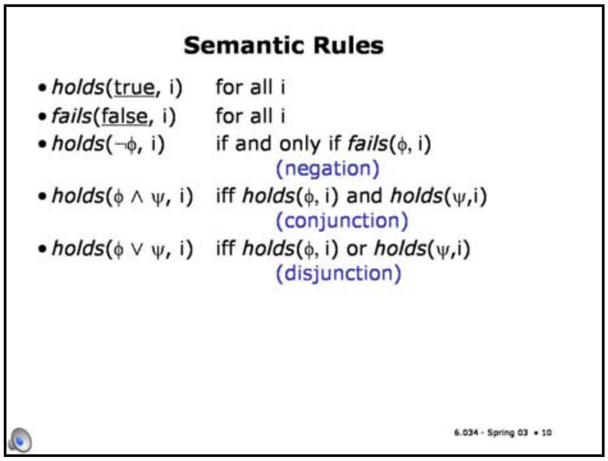
Slide 9.2.9
When is Phi "wedge" Psi true in an interpretation i? Whenever both Phi and Psi are true in i. This is called "conjunction". And we'll start calling that symbol "and" instead of "wedge", now that we know what it means.



Semantic Rules

- *holds*(<u>true</u>, i) for all i
- *fails*(<u>false</u>, i) for all i
- *holds*($\neg\phi$, i) if and only if *fails*(ϕ, i)
  (negation)
- *holds*($\phi \wedge \psi$, i) iff *holds*(ϕ, i) and *holds*(ψ,i)
  (conjunction)

6.034 - Spring 03 • 9



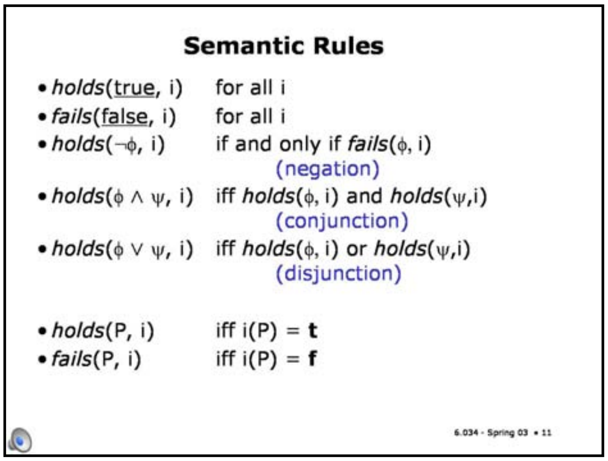
Semantic Rules

- *holds*(<u>true</u>, i) for all i
- *fails*(<u>false</u>, i) for all i
- *holds*($\neg\phi$, i) if and only if *fails*(ϕ, i)
  (negation)
- *holds*($\phi \wedge \psi$, i) iff *holds*(ϕ, i) and *holds*(ψ,i)
  (conjunction)
- *holds*($\phi \vee \psi$, i) iff *holds*(ϕ, i) or *holds*(ψ,i)
  (disjunction)

6.034 - Spring 03 • 10

Slide 9.2.10
When is Phi "vee" Psi true in an interpretation i? Whenever either Phi or Psi is true in i. This is called "disjuction", and we'll call the "vee" symbol "or". It is not an exclusive or; so that if both Phi and Psi are true in i, then Phi "vee" Psi is also true in i.

Slide 9.2.11
Now we have one more clause in our definition. I'm going to do it by example. Imagine that we have a sentence P. P is one of our propositional variables. How do we know whether it is true in interpretation i? Well, since i is a mapping from variables to truth values, I can simply look P up in i and return whatever truth value was assigned to P by i.

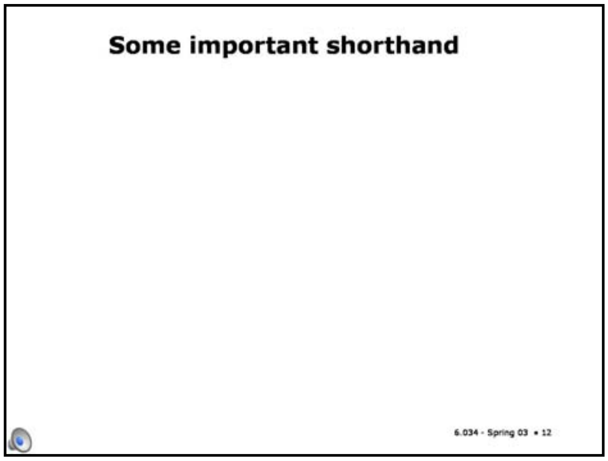


Semantic Rules

- *holds*($\underline{\text{true}}$, i) for all i
- *fails*($\underline{\text{false}}$, i) for all i
- *holds*($\neg\phi$, i) if and only if *fails*(ϕ, i) (negation)
- *holds*($\phi \wedge \psi$, i) iff *holds*(ϕ, i) and *holds*(ψ,i) (conjunction)
- *holds*($\phi \vee \psi$, i) iff *holds*(ϕ, i) or *holds*(ψ,i) (disjunction)

- *holds*(P, i) iff i(P) = **t**
- *fails*(P, i) iff i(P) = **f**

6.034 - Spring 03 • 11

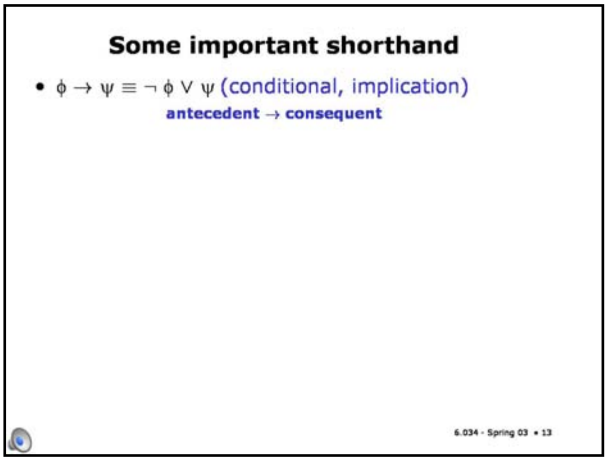


Some important shorthand

6.034 - Spring 03 • 12

Slide 9.2.12
It seems like we left out the arrows in the semantic definitions of the previous slide. But the arrows are not strictly necessary; that is, it's going to turn out that you can say anything you want to without them, but they're a convenient shorthand. (In fact, you can also do without either "or" or "and", but we'll see that later).

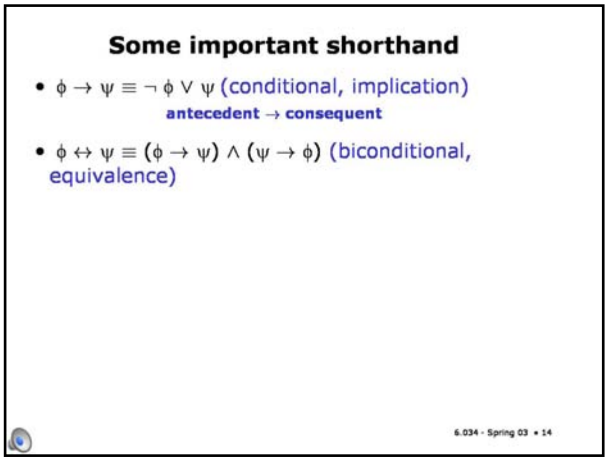
Slide 9.2.13
So, we can define Phi "arrow" Psi as being equivalent to not Phi or Psi. That is, no matter what Phi and Psi are, and in every interpretation, (Phi "arrow" Psi) will have the same truth value as (not Phi or Psi). We will now call this arrow relationship "implication". We'll say that Phi implies Psi. We may also call this a conditional expression: Psi is true if Phi is true. In such a statement, Phi is referred to as the antecedent and Psi as the consequent.



Some important shorthand

- $\phi \rightarrow \psi \equiv \neg \phi \vee \psi$ (conditional, implication)
  antecedent → consequent

6.034 - Spring 03 • 13



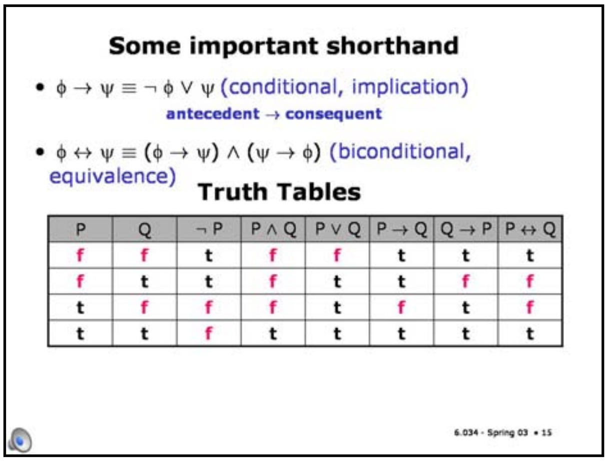
Some important shorthand

- $\phi \rightarrow \psi \equiv \neg \phi \vee \psi$ (conditional, implication)
  antecedent → consequent
- $\phi \leftrightarrow \psi \equiv (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$ (biconditional, equivalence)

6.034 - Spring 03 • 14

Slide 9.2.14
Finally, the double arrow just means that we have single arrows going both ways. This is sometimes called a "bi-conditional" or "equivalence" statement. It means that in every interpretation, Phi and Psi have the same truth value.

Slide 9.2.15
Just so you can see how all of these operators work, here are the truth tables. Consider a world with two propositional variables, P and Q. There are four possible interpretations in such a world (one for every combination of assignments to the variables; in general, in a world with n variables, there will be 2^n possible interpretations). Each row of the truth table corresponds to a possible interpretation, and we've filled in the values it assigns to P and Q in the first two columns. Once we have chosen an interpretation (a row in the table), then the semantic rules tell us exactly what the truth value of every single legal sentence must be. Here we show the truth values for six different sentences made up from P and Q.



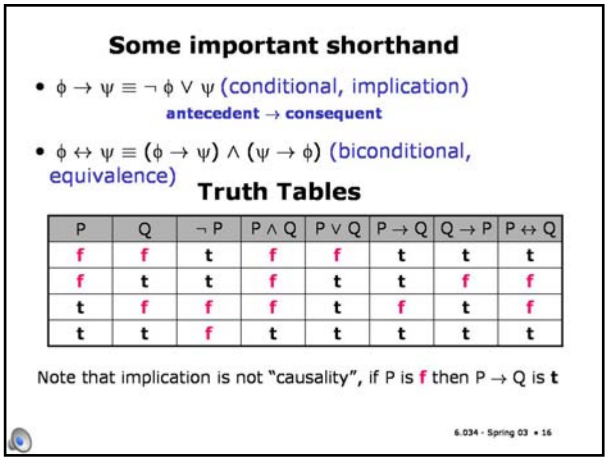
Some important shorthand

- $\phi \rightarrow \psi \equiv \neg \phi \vee \psi$ (conditional, implication)
  antecedent → consequent
- $\phi \leftrightarrow \psi \equiv (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$ (biconditional, equivalence)

Truth Tables

P	Q	¬ P	P ∧ Q	P ∨ Q	P → Q	Q → P	P ↔ Q
f	f	t	f	f	t	t	t
f	t	t	f	t	t	f	f
t	f	f	f	t	f	t	f
t	t	f	t	t	t	t	t

6.034 - Spring 03 • 15



Some important shorthand

- $\phi \rightarrow \psi \equiv \neg \phi \vee \psi$ (conditional, implication)
  antecedent → consequent
- $\phi \leftrightarrow \psi \equiv (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$ (biconditional, equivalence)

Truth Tables

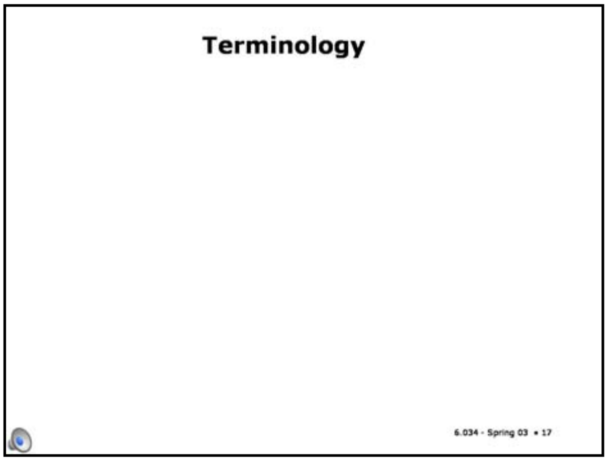
P	Q	¬ P	P ∧ Q	P ∨ Q	P → Q	Q → P	P ↔ Q
f	f	t	f	f	t	t	t
f	t	t	f	t	t	f	f
t	f	f	f	t	f	t	f
t	t	f	t	t	t	t	t

Note that implication is not "causality", if P is **f** then P → Q is **t**

6.034 - Spring 03 • 16

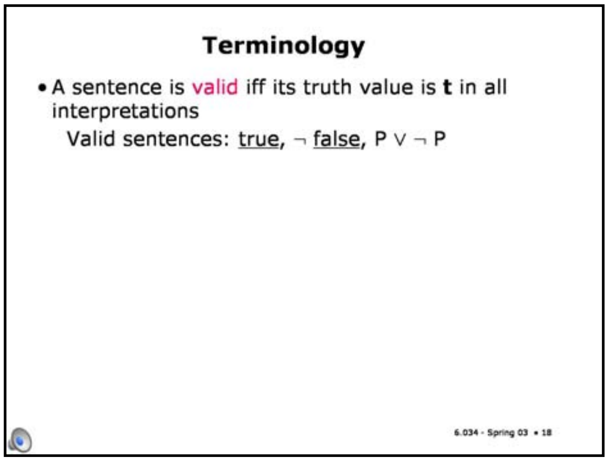
Slide 9.2.16
Most of them are fairly obvious, but it's worth studying the truth table for implication fairly closely. In particular, note that (P implies Q) is true whenever P is false. You can see that this is reasonable by thinking about an English sentence like "If pigs can fly then ...". Once you start with a false condition, you can finish with anything, and the sentence will be true. Implication doesn't mean "causes". It doesn't mean "is related" in any kind of real-world way; it is just a bare, formal definition of not P or Q.

Slide 9.2.17
Now we'll define some terminology on this slide and the next, then do a lot of examples.



Terminology

6.034 - Spring 03 • 17



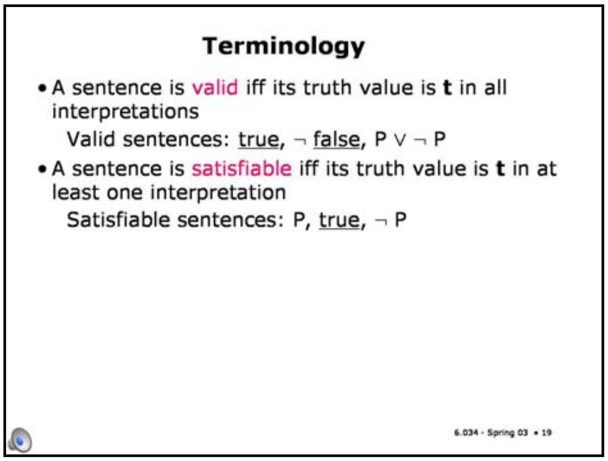
Terminology

- A sentence is valid iff its truth value is **t** in all interpretations
  Valid sentences: <u>true</u>, ¬ <u>false</u>, P ∨ ¬ P

6.034 - Spring 03 • 18

Slide 9.2.18
A sentence is **valid** if and only if it is true in all interpretations. We have already seen one example of a valid sentence. What was it? True. Another one is "not false". A more interesting one is "P or not P". No matter what truth value is assigned to P by the interpretation, "P or not P" is true.

Slide 9.2.19
A sentence is satisfiable if and only if it's true in at least one interpretation. The sentence P is satisfiable. The sentence True is satisfiable. Not P is satisfiable.

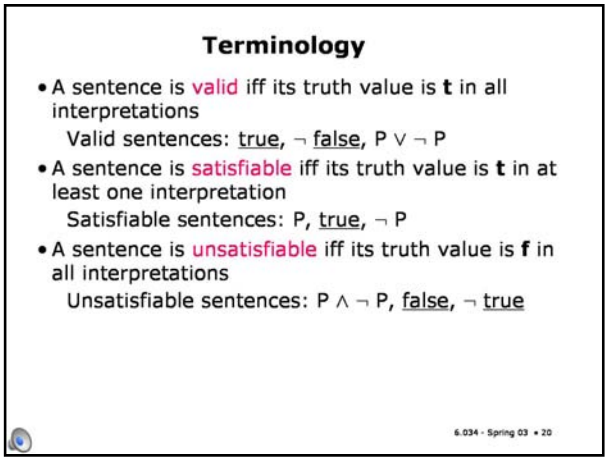


Terminology

- A sentence is valid iff its truth value is **t** in all interpretations
  Valid sentences: true, ¬ false, P ∨ ¬ P
- A sentence is satisfiable iff its truth value is **t** in at least one interpretation
  Satisfiable sentences: P, true, ¬ P

6.034 - Spring 03 • 19

Slide 9.2.20
A sentence is unsatisfiable if and only if it's false in every interpretation. Some unsatisfiable sentences are: false, not true, P and not P.

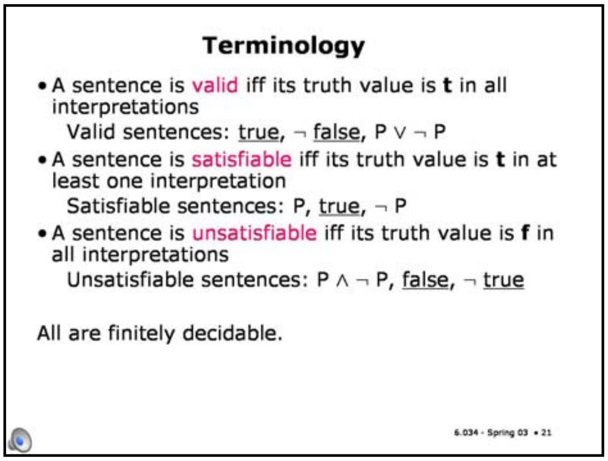


Terminology

- A sentence is valid iff its truth value is **t** in all interpretations
  Valid sentences: true, ¬ false, P ∨ ¬ P
- A sentence is satisfiable iff its truth value is **t** in at least one interpretation
  Satisfiable sentences: P, true, ¬ P
- A sentence is unsatisfiable iff its truth value is **f** in all interpretations
  Unsatisfiable sentences: P ∧ ¬ P, false, ¬ true

6.034 - Spring 03 • 20

Slide 9.2.21
We can use the method of truth tables to check these things. If I wanted to know whether a particular sentence was valid, or if I wanted to know if it was satisfiable or unsatisfiable, I could just make a truth table. I'd write down all the interpretations, figure out the value of the sentence in each interpretation, and if they're all true, it's valid. If they're all false, it's unsatisfiable. If it's somewhere in between, it's satisfiable. So there's a reliable way; there's a completely dopey, tedious, mechanical way to figure out if a sentence is has one of these properties. That's not true in all logics. This is a useful, special property of propositional logic. It might take you a lot of time, but it's a finite amount of time and you can decide any of these questions.



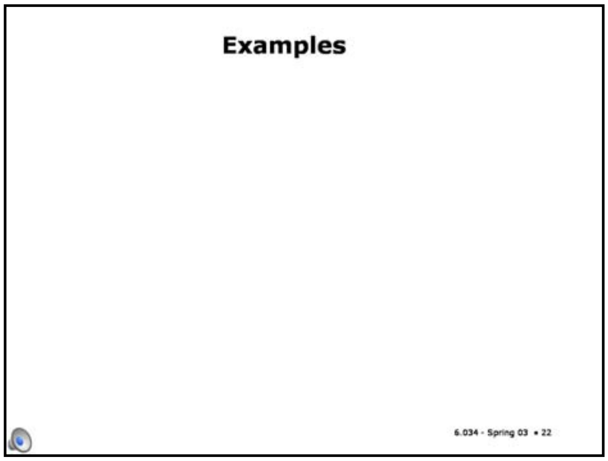
Terminology

- A sentence is valid iff its truth value is **t** in all interpretations
  Valid sentences: true, ¬ false, P ∨ ¬ P
- A sentence is satisfiable iff its truth value is **t** in at least one interpretation
  Satisfiable sentences: P, true, ¬ P
- A sentence is unsatisfiable iff its truth value is **f** in all interpretations
  Unsatisfiable sentences: P ∧ ¬ P, false, ¬ true

All are finitely decidable.

6.034 - Spring 03 • 21

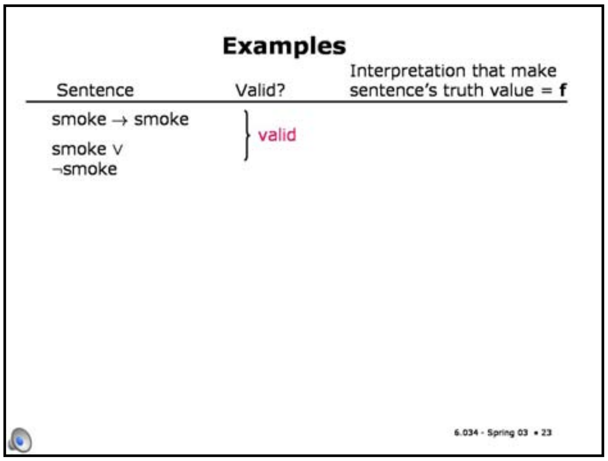
Slide 9.2.22
Let's work through some examples. We can think about whether they're valid or unsatisfiable or satisfiable.



Examples

6.034 - Spring 03 • 22

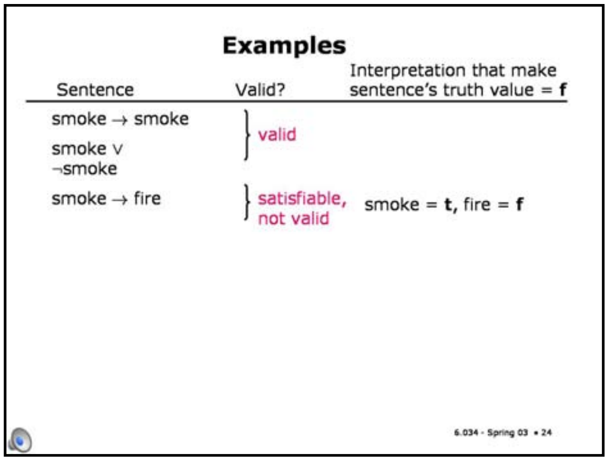
Slide 9.2.23
What about "smoke implies smoke"? Rather than doing a whole truth table it might be easier if we can convert it into smoke or not smoke, right? The definition of A implies B is not A or B. And we said that smoke or not smoke was valid already.



Examples

Sentence	Valid?	Interpretation that make sentence's truth value = **f**
smoke → smoke smoke ∨ ¬smoke	valid	

6.034 - Spring 03 • 23



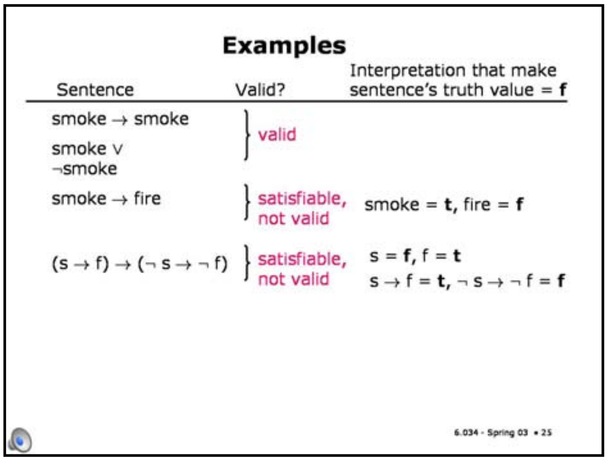
Examples

Sentence	Valid?	Interpretation that make sentence's truth value = **f**
smoke → smoke smoke ∨ ¬smoke	valid	
smoke → fire	satisfiable, not valid	smoke = **t**, fire = **f**

6.034 - Spring 03 • 24

Slide 9.2.24
What about "smoke implies fire"? It's satisfiable, because there's an interpretation of these two symbols that makes it true. There are other interpretations that make it false. I should say, everything that's valid is also satisfiable.

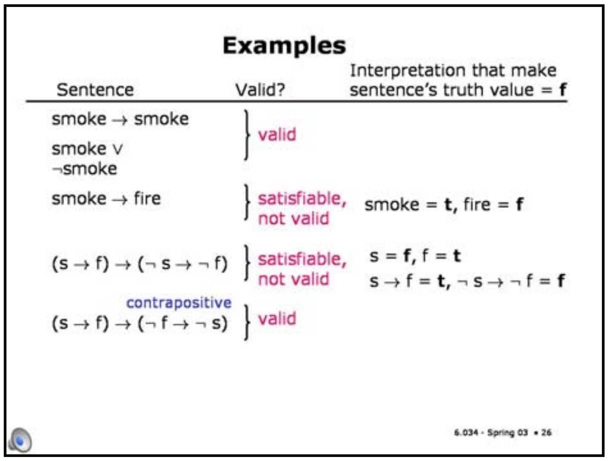
Slide 9.2.25
Here is a form of reasoning that you hear people do a lot, but the question is, is it okay? "Smoke implies fire implies not smoke implies not fire." It's invalid. We could show that by drawing out the truth table (and you should do it as an exercise if the answer is not obvious to you). Another way to show that a sentence is not valid is to give an interpretation that makes the sentence have the truth value **f**. In this case, if we give "smoke" the truth value **f** and and "fire" the truth value **t**, then the whole sentence has truth value **f**.



Examples

Sentence	Valid?	Interpretation that make sentence's truth value = **f**
smoke → smoke smoke ∨ ¬smoke	valid	
smoke → fire	satisfiable, not valid	smoke = **t**, fire = **f**
(s → f) → (¬ s → ¬ f)	satisfiable, not valid	s = **f**, f = **t** s → f = **t**, ¬ s → ¬ f = **f**

6.034 - Spring 03 • 25



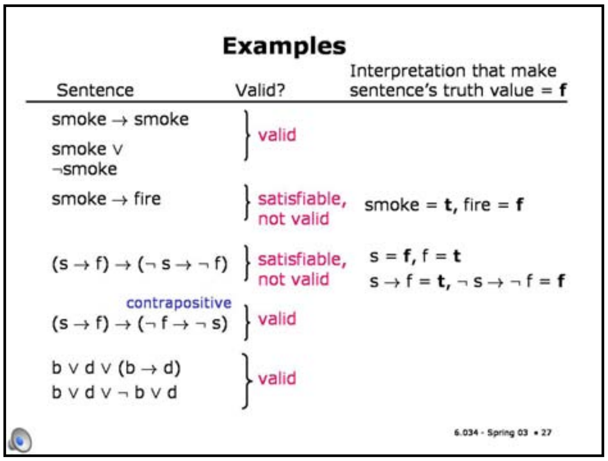
Examples

Sentence	Valid?	Interpretation that make sentence's truth value = **f**
smoke → smoke smoke ∨ ¬smoke	valid	
smoke → fire	satisfiable, not valid	smoke = **t**, fire = **f**
(s → f) → (¬ s → ¬ f)	satisfiable, not valid	s = **f**, f = **t** s → f = **t**, ¬ s → ¬ f = **f**
contrapositive (s → f) → (¬ f → ¬ s)	valid	

6.034 - Spring 03 • 26

Slide 9.2.26
Reasoning in the other direction is okay, though. So the sentence "smoke implies fire implies not fire implies not smoke" is valid. And for those of you who love terminology, this thing is called the contrapositive. So, if there's no fire, then there's no smoke.

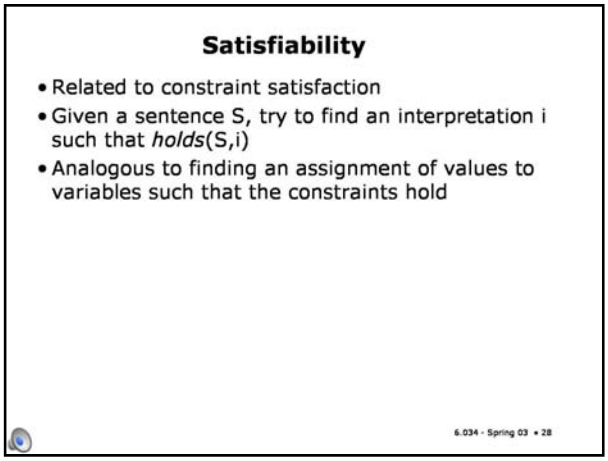
Slide 9.2.27
What about "b or d or (b implies d)"? We can rewrite that (using the definition of implication) into "b or d or not b or d", which is valid, because in every interpretation either b or not b must be true.



Examples

Sentence	Valid?	Interpretation that make sentence's truth value = **f**
smoke → smoke smoke ∨ ¬smoke	valid	
smoke → fire	satisfiable, not valid	smoke = **t**, fire = **f**
(s → f) → (¬ s → ¬ f)	satisfiable, not valid	s = **f**, f = **t** s → f = **t**, ¬ s → ¬ f = **f**
contrapositive (s → f) → (¬ f → ¬ s)	valid	
b ∨ d ∨ (b → d) b ∨ d ∨ ¬ b ∨ d	valid	

6.034 - Spring 03 • 27



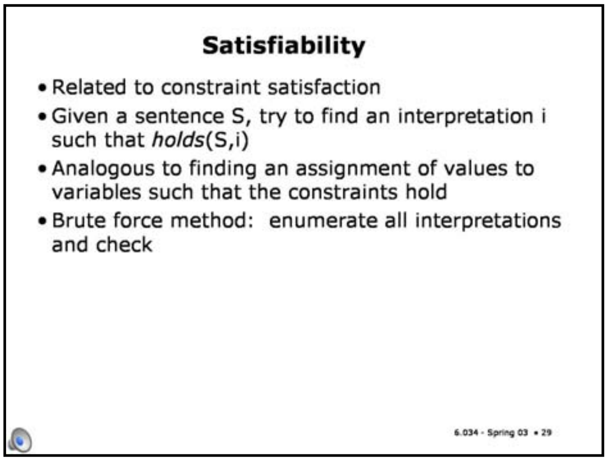
Satisfiability

- Related to constraint satisfaction
- Given a sentence S, try to find an interpretation i such that *holds*(S,i)
- Analogous to finding an assignment of values to variables such that the constraints hold

6.034 - Spring 03 • 28

Slide 9.2.28
The problem of deciding whether a sentence is satisfiable is related to constraint satisfaction: you have to find an interpetation i such that the sentence holds in that interpretation. That's analogous to finding an assignment of values to variables so that the constraints are satisfied.

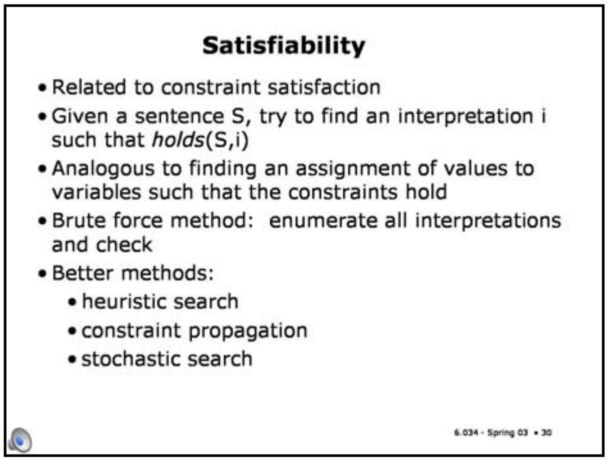
Slide 9.2.29
We could try to solve these problems using the brute-force method of enumerating all possible interpretations, then looking for one that makes the sentence true.



Satisfiability

- Related to constraint satisfaction
- Given a sentence S, try to find an interpretation i such that *holds*(S,i)
- Analogous to finding an assignment of values to variables such that the constraints hold
- Brute force method: enumerate all interpretations and check

6.034 - Spring 03 • 29



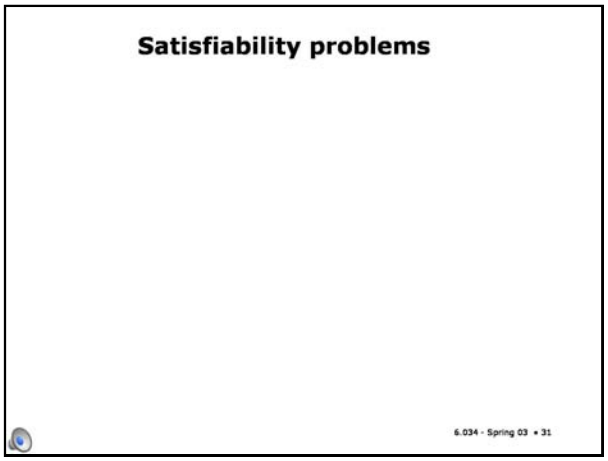
Satisfiability

- Related to constraint satisfaction
- Given a sentence S, try to find an interpretation i such that *holds*(S,i)
- Analogous to finding an assignment of values to variables such that the constraints hold
- Brute force method: enumerate all interpretations and check
- Better methods:
  - heuristic search
  - constraint propagation
  - stochastic search

6.034 - Spring 03 • 30

Slide 9.2.30
Better would be to use methods from constraint satisfaction. There are a number of search algorithms that have been specially adapted to solving satisfiability problems as quickly as possible, using combinations of backtracking, constraint propagation, and variable ordering.

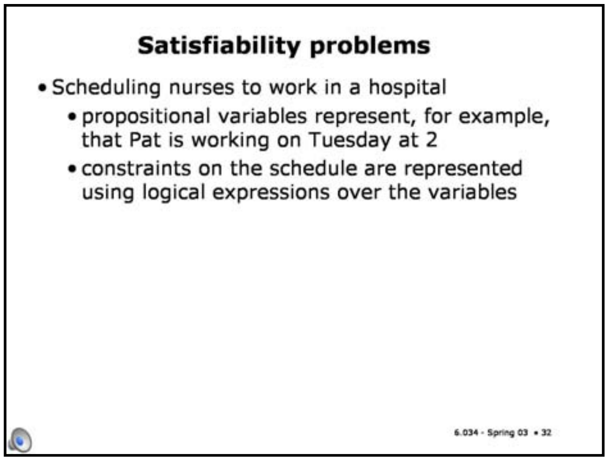
Slide 9.2.31
There are lots of satisfiability problems in the real world. They end up being expressed essentially as lists of boolean logic expressions, where you're trying to find some assignment of values to variables that makes the sentence true.



Satisfiability problems

6.034 - Spring 03 • 31

Slide 9.2.32
One example is scheduling nurses to work shifts in a hospital. Different people have different constraints, some don't want to work at night, no individual can work more than this many hours out of that many hours, these two people don't want to be on the same shift, you have to have at least this many per shift and so on. So you can often describe a setting like that as a bunch of constraints on a set of variables.



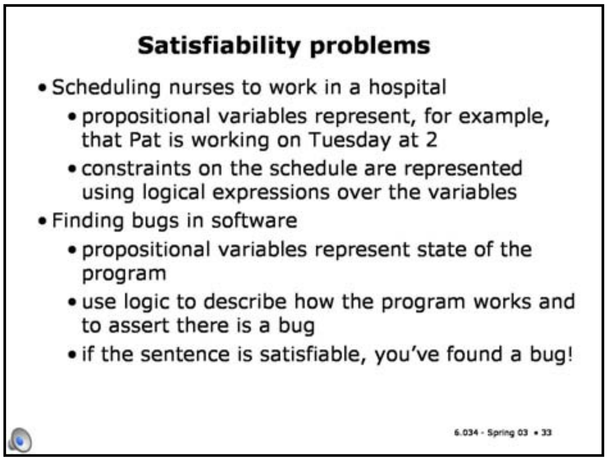
Satisfiability problems

- Scheduling nurses to work in a hospital
  - propositional variables represent, for example, that Pat is working on Tuesday at 2
  - constraints on the schedule are represented using logical expressions over the variables

6.034 - Spring 03 • 32

Slide 9.2.33
There's an interesting application of satisfiability that's going on here at MIT in the Lab for Computer Science. Professor Daniel Jackson's interested in trying to find bugs in programs. That's a good thing to do, but (as you know!) it's hard for humans to do reliably, so he wants to get the computer to do it automatically.

One way to do it is to essentially make a small example instance of a program. So an example of a kind of program that he might want to try to find a bug in would be an air traffic controller. The air traffic controller has rules that specify how it works. So you could write down the logical specification of how the air traffic control protocol works, and then you could write down another sentence that says, "and there are two airplanes on the same runway at the same time." And then you could see if there is a satisfying assignment; whether there is a configuration of airplanes and things that actually satisfies the specifications of the air traffic control protocol and also has two airplanes on the same runway at the same time. And if you can find one -- if that whole sentence is satisfiable, then you have a problem in your air traffic control protocol.



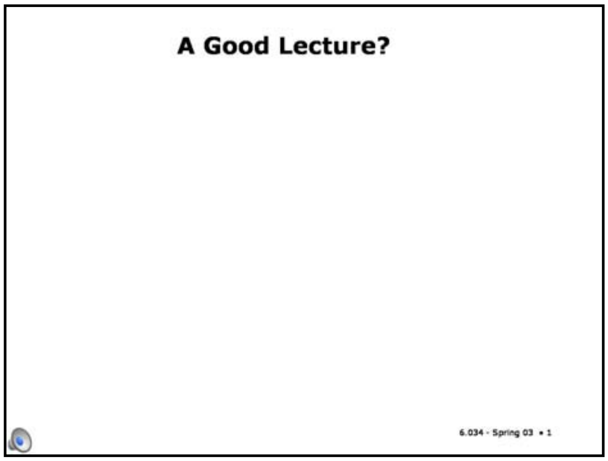
Satisfiability problems

- Scheduling nurses to work in a hospital
  - propositional variables represent, for example, that Pat is working on Tuesday at 2
  - constraints on the schedule are represented using logical expressions over the variables
- Finding bugs in software
  - propositional variables represent state of the program
  - use logic to describe how the program works and to assert there is a bug
  - if the sentence is satisfiable, you've found a bug!

6.034 - Spring 03 • 33

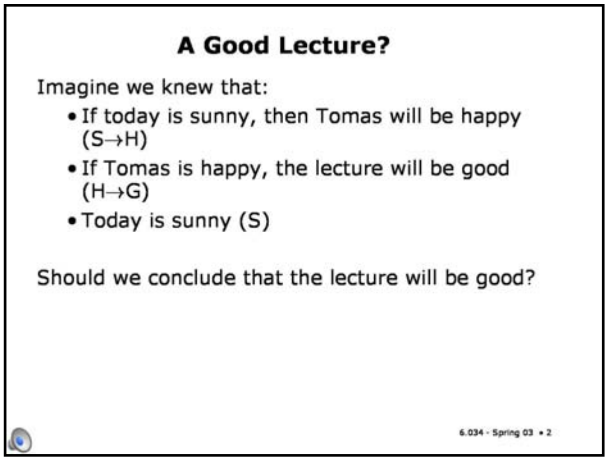
6.034 Notes: Section 9.3

Slide 9.3.1
One reason for writing down logical descriptions of situations is that they will allow us to draw conclusions about other aspects of the situation we've described.



A Good Lecture?

6.034 - Spring 03 • 1



A Good Lecture?

Imagine we knew that:

- If today is sunny, then Tomas will be happy (S→H)
- If Tomas is happy, the lecture will be good (H→G)
- Today is sunny (S)

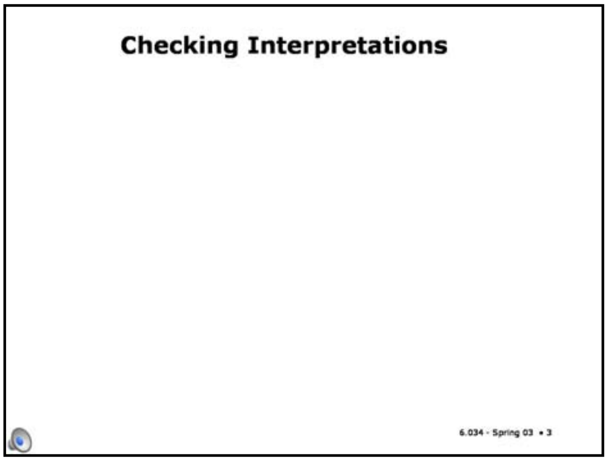
Should we conclude that the lecture will be good?

6.034 - Spring 03 • 2

Slide 9.3.2
Imagine that we knew the following things to be true: If today is sunny, Tomas will be happy; if Tomas is happy, the lecture will be good; and today is sunny.

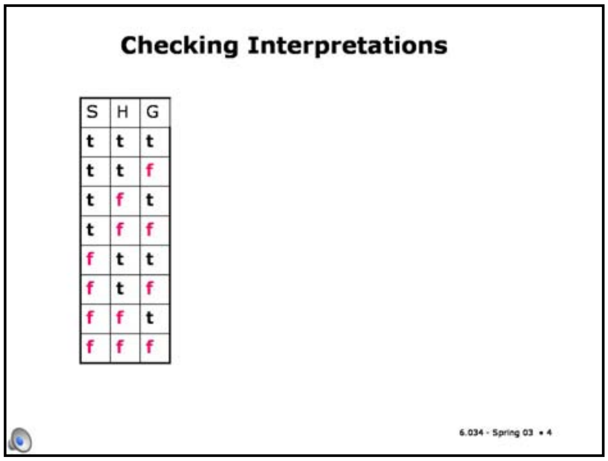
Does this mean that the lecture will be good?

Slide 9.3.3
One way to think about this is to start by figuring out what set of interpretations make our original sentences true. Then, if G is true in all those interpretations, it must be okay to conclude it from the sentences we started out with (sometimes called our knowledge base).



Checking Interpretations

6.034 - Spring 03 • 3



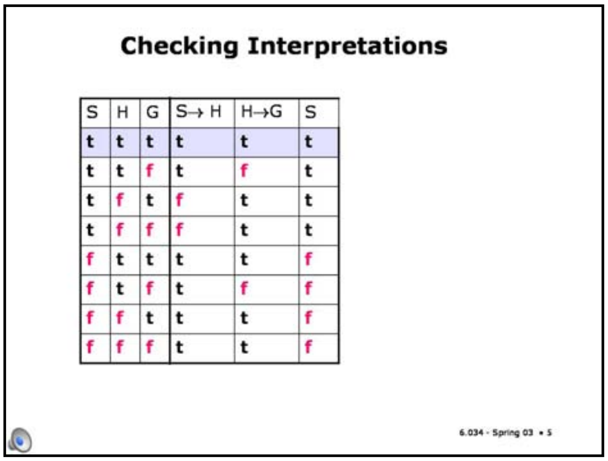
Checking Interpretations

S	H	G
t	t	t
t	t	f
t	f	t
t	f	f
f	t	t
f	t	f
f	f	t
f	f	f

6.034 - Spring 03 • 4

Slide 9.3.4
In a universe with only three variables, there are 8 possible interpretations in total.

Slide 9.3.5
Only one of these interpretations makes all the sentences in our knowledge base true: S = true, H = true, G = true.

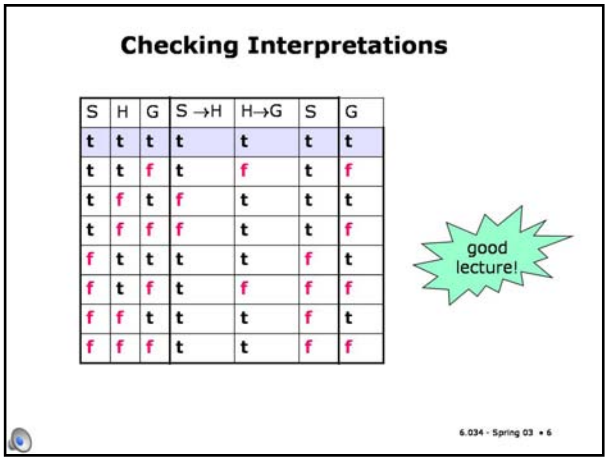


Checking Interpretations

S	H	G	S→ H	H→G	S
t	t	t	t	t	t
t	t	f	t	f	t
t	f	t	f	t	t
t	f	f	f	t	t
f	t	t	t	t	f
f	t	f	t	f	f
f	f	t	t	t	f
f	f	f	t	t	f

6.034 - Spring 03 • 5

Slide 9.3.6
And it's easy enough to check that G is true in that interpretation, so it seems like it must be reasonable to draw the conclusion that the lecture will be good. (Good thing!).



Checking Interpretations

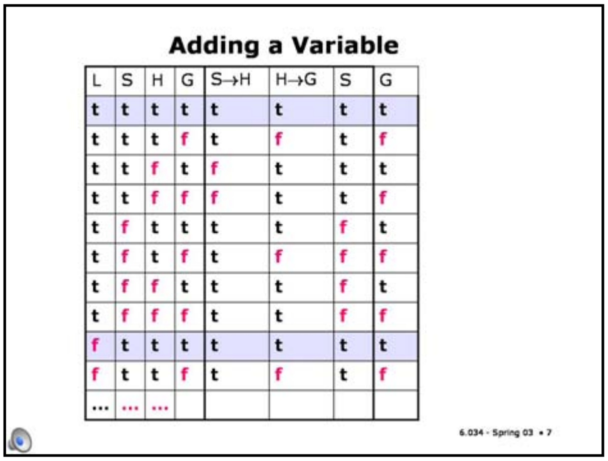
S	H	G	S →H	H→G	S	G
t	t	t	t	t	t	t
t	t	f	t	f	t	f
t	f	t	f	t	t	t
t	f	f	f	t	t	f
f	t	t	t	t	f	t
f	t	f	t	f	f	f
f	f	t	t	t	f	t
f	f	f	t	t	f	f

good lecture!

6.034 - Spring 03 • 6

Slide 9.3.7
If we added another variable to our domain, say whether Leslie is happy (L), then we'd have two interpretations that satisfy the KB: S = true, H = true, G = true, L = true; and S = true, H = true, G = true, L = false.

G is true in both of these interpretations, so, again, if the KB is true, then G must also be true.

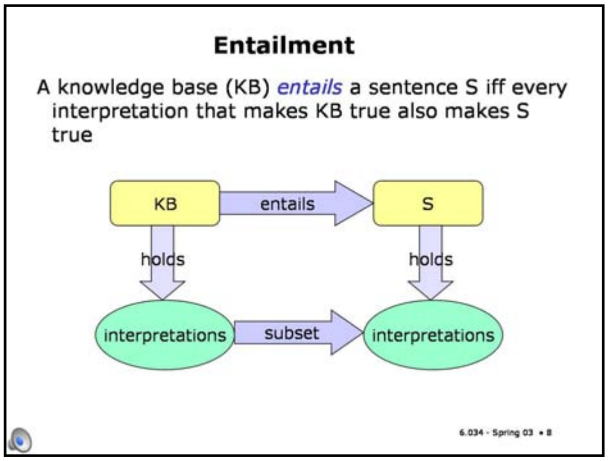


Adding a Variable

L	S	H	G	S→H	H→G	S	G
t	t	t	t	t	t	t	t
t	t	t	f	t	f	t	f
t	t	f	t	f	t	t	t
t	t	f	f	f	t	t	f
t	f	t	t	t	t	f	t
t	f	t	f	t	f	f	f
t	f	f	t	t	t	f	t
t	f	f	f	t	t	f	f
f	t	t	t	t	t	t	t
f	t	t	f	t	f	t	f
...	...	...					

6.034 - Spring 03 • 7

Slide 9.3.8
There is a general idea called "entailment" that signifies a relationship between a knowledge base and another sentence. If whenever the KB is true, the conclusion has to be true (that is, if every interpretation that satisfies the KB also satisfies the conclusion), we'll say that the KB "entails" the conclusion. You can think of entailment as something like "follows from", or "it's okay to conclude from".

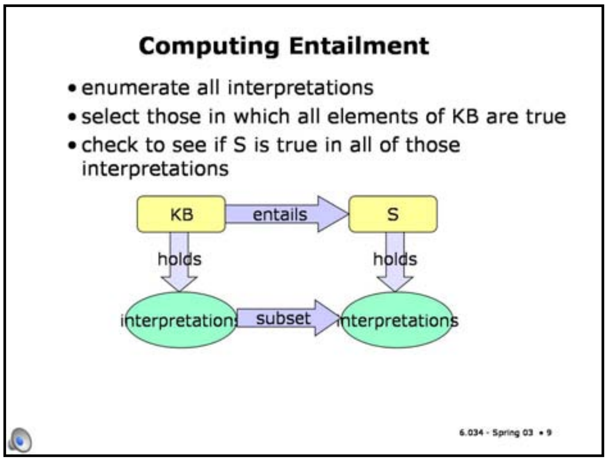


Entailment

A knowledge base (KB) *entails* a sentence S iff every interpretation that makes KB true also makes S true

6.034 - Spring 03 • 8

Slide 9.3.9
The method of enumerating all the interpretations that satisfy the KB, and then checking to see if the conclusion is true in all of them is a correct way to test entailment.

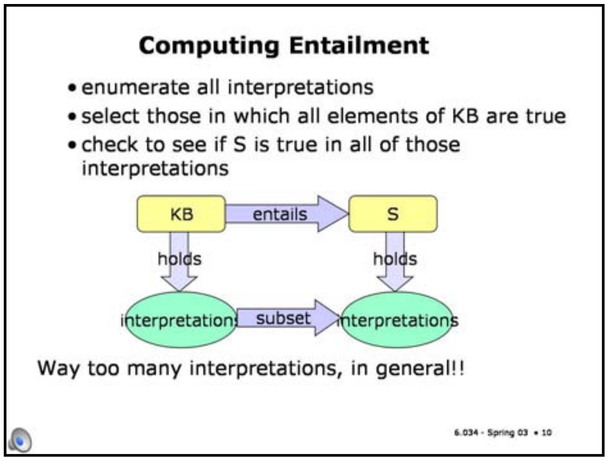


Computing Entailment

- enumerate all interpretations
- select those in which all elements of KB are true
- check to see if S is true in all of those interpretations

KB entails S
holds
holds
interpretations subset interpretations

6.034 - Spring 03 • 9



Computing Entailment

- enumerate all interpretations
- select those in which all elements of KB are true
- check to see if S is true in all of those interpretations

KB entails S
holds
holds
interpretations subset interpretations

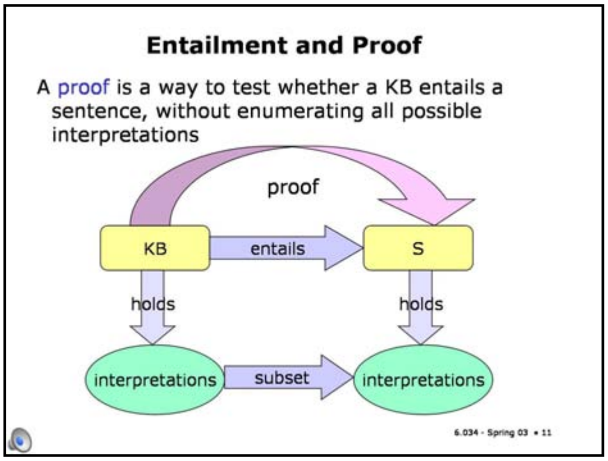
Way too many interpretations, in general!!

6.034 - Spring 03 • 10

Slide 9.3.10
But now, what if we were to add 6 more propositional variables to our domain? Then we'd have 2^10 = 1024 interpretations to check, which is way too much work to do (and, in the first order case, we'll find that we might have infinitely many intepretations, which is definitely too much work to enumerate!!).

Slide 9.3.11
So what we'd really like is a way to figure out whether a KB entails a conclusion without enumerating all of the possible interpretations.

A proof is a way to test whether a KB entails a sentence, without enumerating all possible interpretations. You can think of it as a kind of shortcut arrow that works directly with the syntactic representations of the KB and the conclusion, without going into the semantic world of interpretations.

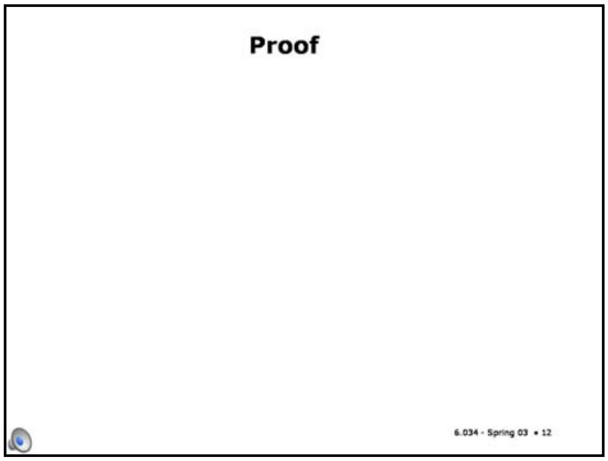


Entailment and Proof

A proof is a way to test whether a KB entails a sentence, without enumerating all possible interpretations

proof
KB entails S
holds
holds
interpretations subset interpretations

6.034 - Spring 03 • 11



Proof

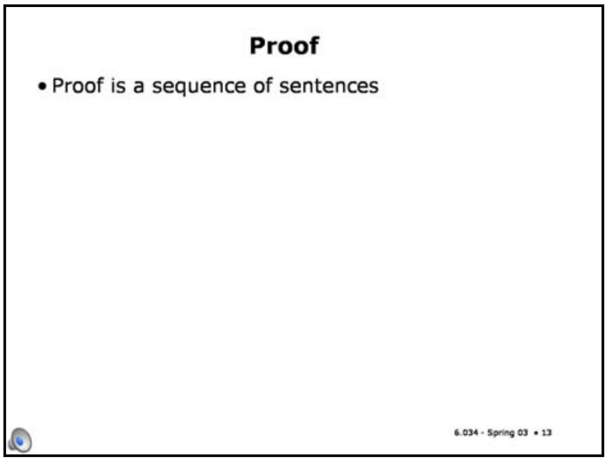
6.034 - Spring 03 • 12

Slide 9.3.12
So what is a proof system? Well, presumably all of you have studied high-school geometry; that's often people's only exposure to formal proof. Remember that? You knew some things about the sides and angles of two triangles and then you applied the side-angle-side theorem to conclude -- at least people in American high schools were familiar with side-angle-side -- The side-angle-side theorem allowed you to conclude that the two triangles were similar, right?

That is formal proof. You've got some set of rules that you can apply. You've got some things written down on your page, and you grind through, applying the rules that you have to the things that are written down, to write some more stuff down until finally you've written down the things that you wanted to, and then you get to declare victory. That's a proof. There are (at least) two styles of proof system; we're going to talk about one briefly here and then go on to the other one at some length in the next two sections.

Natural deduction refers to a set of proof systems that are very similar to the kind of system you used in high-school geometry. We'll talk a little bit about natural deduction just to give you a flavor of how it goes in propositional logic, but it's going to turn out that it's not very good as a general strategy for computers. It's a proof system that humans like, and then we'll talk about a proof system that computers like, to the extent that computers can like anything.

Slide 9.3.13
A proof is a sequence of sentences. This is going to be true in almost all proof systems.

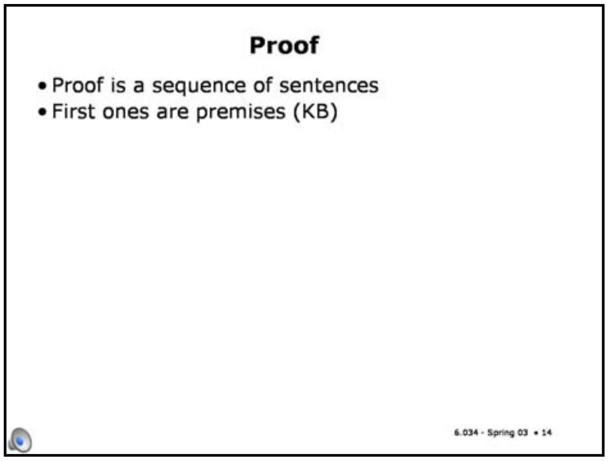


Proof

- Proof is a sequence of sentences

6.034 - Spring 03 • 13

Slide 9.3.14
First we'll list the premises. These are the sentences in your knowledge base. The things that you know to start out with. You're allowed to write those down on your page. Sometimes they're called the "givens." You can put the givens down.

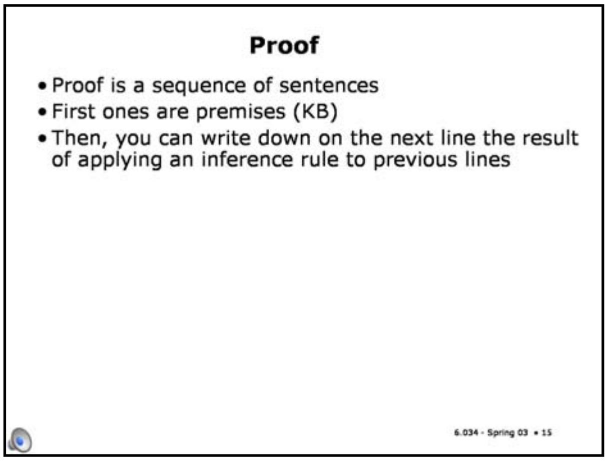


Proof

- Proof is a sequence of sentences
- First ones are premises (KB)

6.034 - Spring 03 • 14

Slide 9.3.15
Then, you can write down on a new line of your proof the results of applying an inference rule to the previous lines.

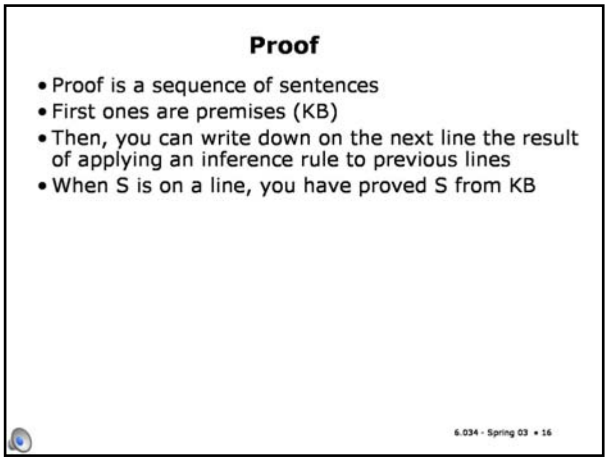


Proof

- Proof is a sequence of sentences
- First ones are premises (KB)
- Then, you can write down on the next line the result of applying an inference rule to previous lines

6.034 - Spring 03 • 15

Slide 9.3.16
Then, when a sentence S is on some line, you have proved S from KB.

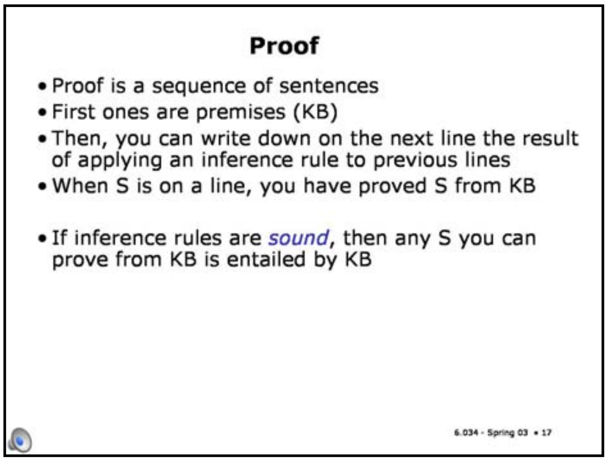


Proof

- Proof is a sequence of sentences
- First ones are premises (KB)
- Then, you can write down on the next line the result of applying an inference rule to previous lines
- When S is on a line, you have proved S from KB

6.034 - Spring 03 • 16

Slide 9.3.17
If your inference rules are **sound**, then any S you can prove from KB is, in fact, entailed by KB. That is, it's legitimate to draw the conclusion S from the assumptions in KB.

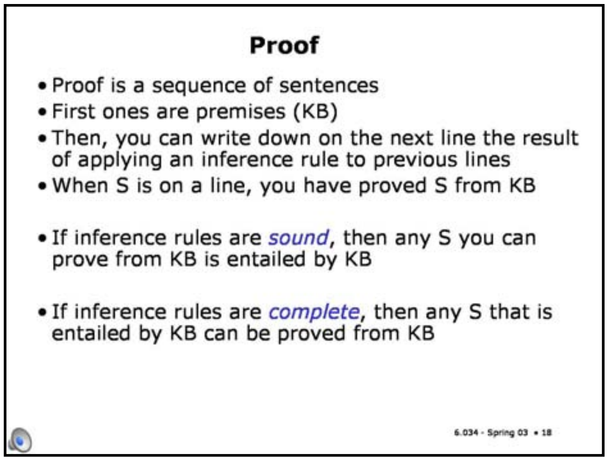


Proof

- Proof is a sequence of sentences
- First ones are premises (KB)
- Then, you can write down on the next line the result of applying an inference rule to previous lines
- When S is on a line, you have proved S from KB

- If inference rules are *sound*, then any S you can prove from KB is entailed by KB

6.034 - Spring 03 • 17



Proof

- Proof is a sequence of sentences
- First ones are premises (KB)
- Then, you can write down on the next line the result of applying an inference rule to previous lines
- When S is on a line, you have proved S from KB

- If inference rules are *sound*, then any S you can prove from KB is entailed by KB

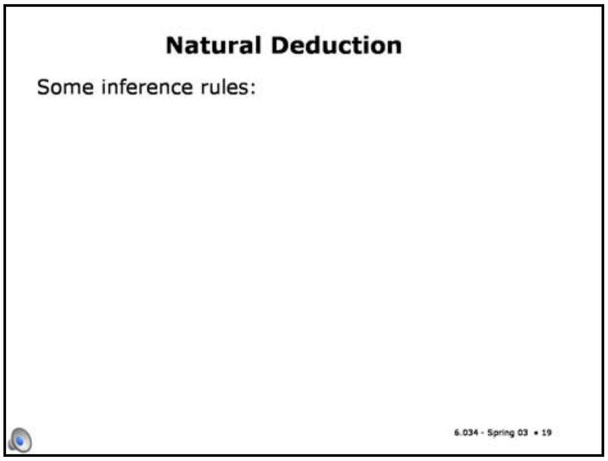
- If inference rules are *complete*, then any S that is entailed by KB can be proved from KB

6.034 - Spring 03 • 18

Slide 9.3.18
If your rules are **complete**, then you can use KB to prove any S that is entailed by the KB. That is, you can prove any legitimate conclusion.

Wouldn't it be great if you were sound and complete derivers of answers to problems? You'd always get an answer and it would always be right!

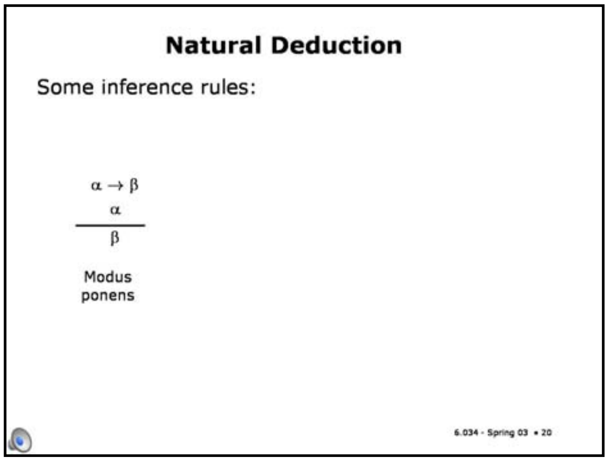
Slide 9.3.19
So let's look at inference rules, and learn how they work by example. We'll look at natural-deduction rules first, because they're easiest to understand.



Natural Deduction

Some inference rules:

6.034 - Spring 03 • 19



Natural Deduction

Some inference rules:

$$\begin{array}{c}\alpha \to \beta \\ \alpha \\ \hline \beta\end{array}$$

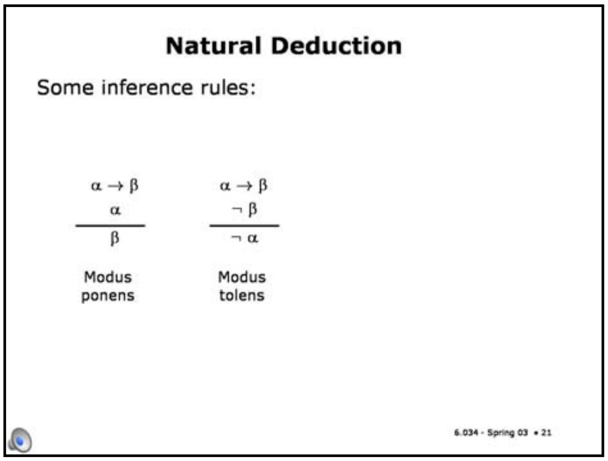
Modus ponens

6.034 - Spring 03 • 20

Slide 9.3.20
Here's a famous one (first written down by Aristotle); it has the great Latin name, "modus ponens", which means "affirming method".

It says that if you have "Alpha implies Beta" written down somewhere on your page, and you have Alpha written down somewhere on your page, then you can write beta down on a new line. (Alpha and Beta here are metavariables, like Phi and Psi, ranging over whole complicated sentences). It's important to remember that inference rules are just about ink on paper, or bits on your computer screen. They're not about anything in the world. Proof is just about writing stuff on a page, just syntax. But if you're careful in your proof rules and they're all sound, then at the end when you have some bit of syntax written down on your page, you can go back via the interpretation to some semantics. So you start out by writing down some facts about the world formally as your knowledge base. You do stuff with ink and paper for a while and now you have some other symbols written down on your page. You can go look them up in the world and say, "Oh, I see. That's what they mean."

Slide 9.3.21
Here's another inference rule. "Modus tollens" (denying method) says that, from "alpha implies beta" and "not beta" you can conclude "not alpha".



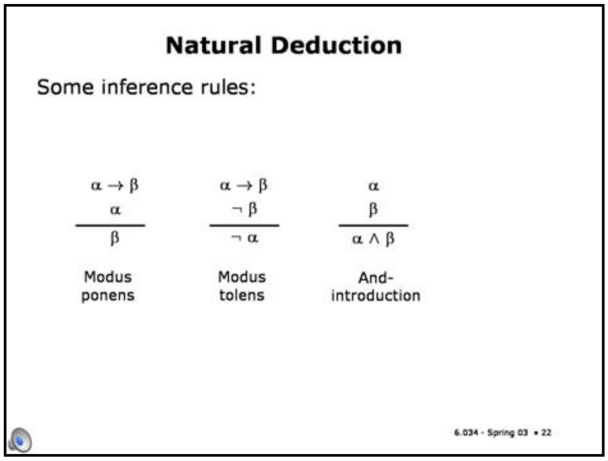
Natural Deduction

Some inference rules:

$$\frac{\begin{array}{c}\alpha \to \beta \\ \alpha\end{array}}{\beta} \qquad \frac{\begin{array}{c}\alpha \to \beta \\ \neg \beta\end{array}}{\neg \alpha}$$

Modus ponens — Modus tolens

6.034 - Spring 03 • 21



Natural Deduction

Some inference rules:

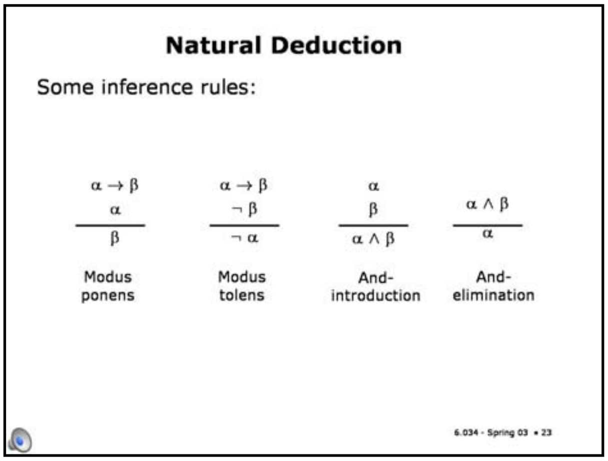
$$\frac{\begin{array}{c}\alpha \to \beta \\ \alpha\end{array}}{\beta} \qquad \frac{\begin{array}{c}\alpha \to \beta \\ \neg \beta\end{array}}{\neg \alpha} \qquad \frac{\begin{array}{c}\alpha \\ \beta\end{array}}{\alpha \wedge \beta}$$

Modus ponens — Modus tolens — And-introduction

6.034 - Spring 03 • 22

Slide 9.3.22
And-introduction say that from "Alpha" and from "Beta" you can conclude "Alpha and Beta". That seems pretty obvious.

Slide 9.3.23
Conversely, and-elimination says that from "Alpha and Beta" you can conclude "Alpha".



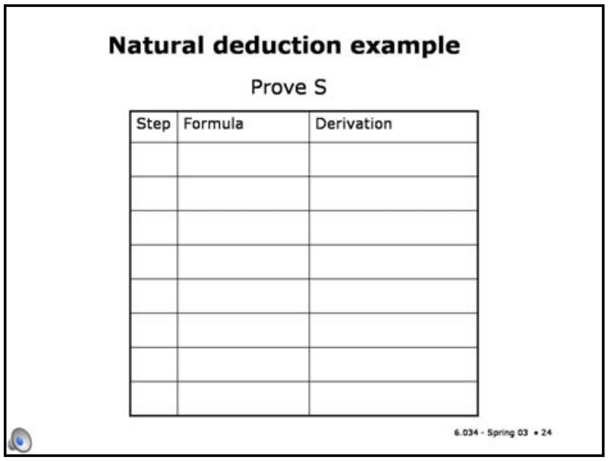
Natural Deduction

Some inference rules:

$$\frac{\begin{array}{c}\alpha \to \beta \\ \alpha\end{array}}{\beta} \qquad \frac{\begin{array}{c}\alpha \to \beta \\ \neg \beta\end{array}}{\neg \alpha} \qquad \frac{\begin{array}{c}\alpha \\ \beta\end{array}}{\alpha \wedge \beta} \qquad \frac{\alpha \wedge \beta}{\alpha}$$

Modus ponens — Modus tolens — And-introduction — And-elimination

6.034 - Spring 03 • 23



Natural deduction example

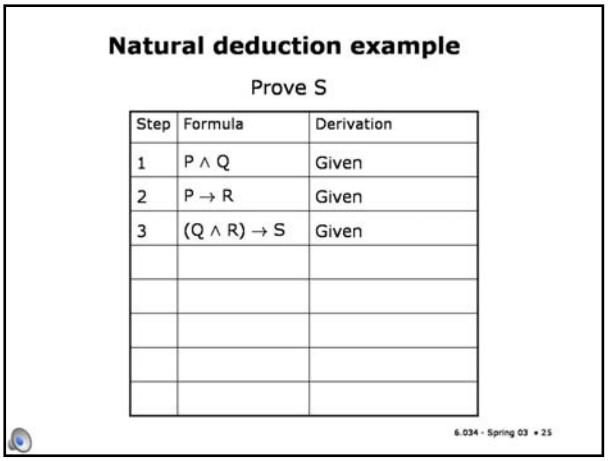
Prove S

Step	Formula	Derivation

6.034 - Spring 03 • 24

Slide 9.3.24
Now let's do a sample proof just to get the idea of how it works. Pretend you're back in high school

Slide 9.3.25
We'll start with 3 sentences in our knowledge base, and we'll write them on the first three lines of our proof: (P and Q), (P implies R), and (Q and R imply S).

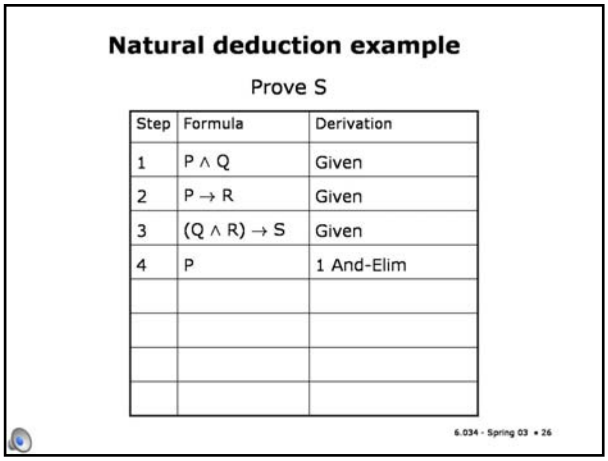


Natural deduction example

Prove S

Step	Formula	Derivation
1	P ∧ Q	Given
2	P → R	Given
3	(Q ∧ R) → S	Given

6.034 - Spring 03 • 25



Natural deduction example

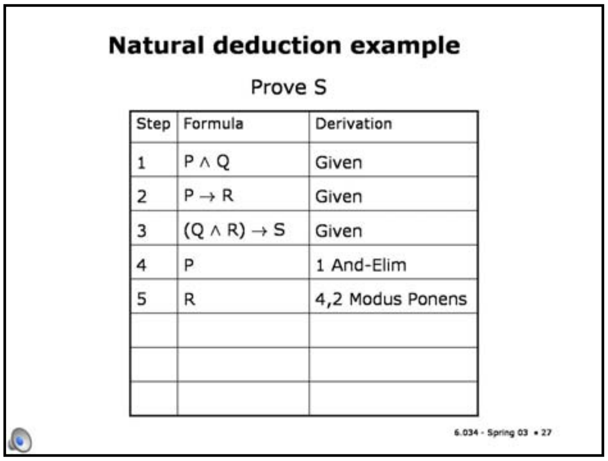
Prove S

Step	Formula	Derivation
1	P ∧ Q	Given
2	P → R	Given
3	(Q ∧ R) → S	Given
4	P	1 And-Elim

6.034 - Spring 03 • 26

Slide 9.3.26
From line 1, using the and-elimination rule, we can conclude P, and write it down on line 4 (together with a reminder of how we derived it).

Slide 9.3.27
From lines 4 and 2, using modus ponens, we can conclude R.

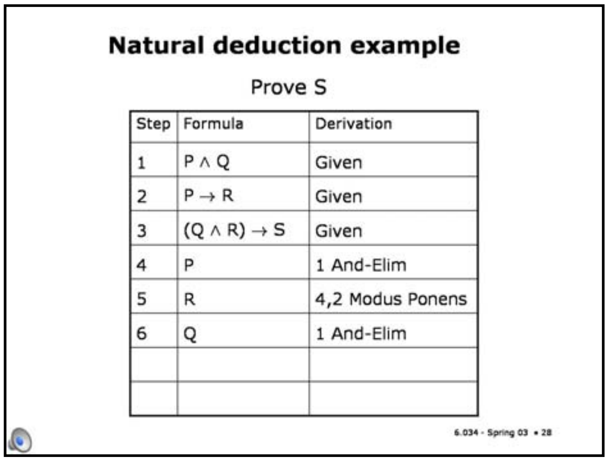


Natural deduction example

Prove S

Step	Formula	Derivation
1	P ∧ Q	Given
2	P → R	Given
3	(Q ∧ R) → S	Given
4	P	1 And-Elim
5	R	4,2 Modus Ponens

6.034 - Spring 03 • 27



Natural deduction example

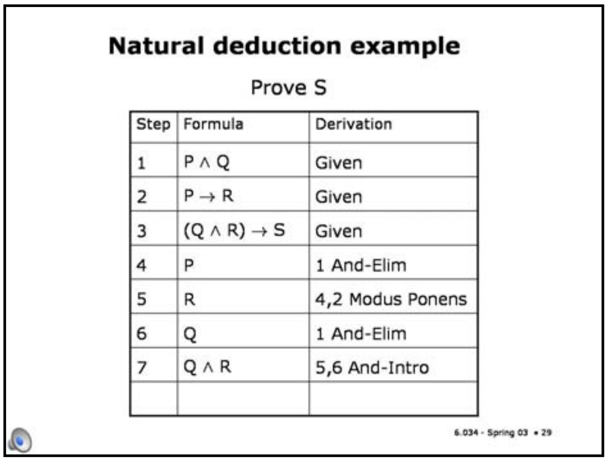
Prove S

Step	Formula	Derivation
1	P ∧ Q	Given
2	P → R	Given
3	(Q ∧ R) → S	Given
4	P	1 And-Elim
5	R	4,2 Modus Ponens
6	Q	1 And-Elim

6.034 - Spring 03 • 28

Slide 9.3.28
From line 1, we can use and-elimination to get Q.

Slide 9.3.29
From lines 5 and 6, we can use and-introduction to get (Q and R).

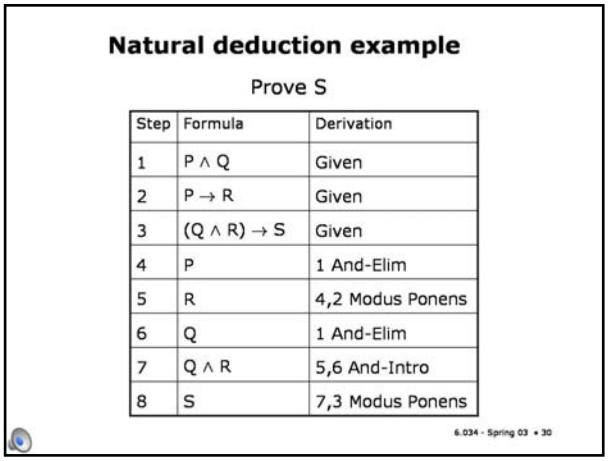


Natural deduction example

Prove S

Step	Formula	Derivation
1	P ∧ Q	Given
2	P → R	Given
3	(Q ∧ R) → S	Given
4	P	1 And-Elim
5	R	4,2 Modus Ponens
6	Q	1 And-Elim
7	Q ∧ R	5,6 And-Intro

6.034 - Spring 03 • 29



Natural deduction example

Prove S

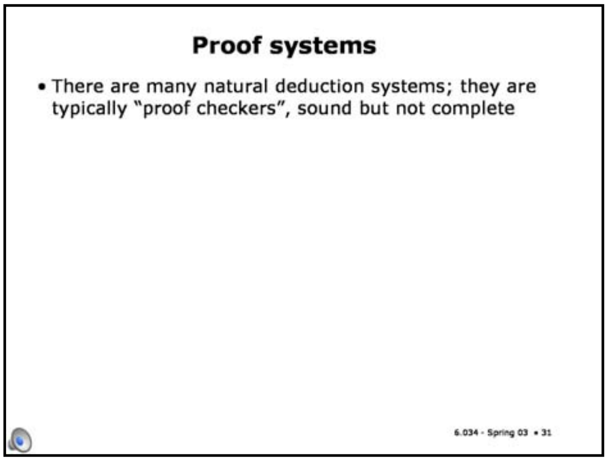
Step	Formula	Derivation
1	P ∧ Q	Given
2	P → R	Given
3	(Q ∧ R) → S	Given
4	P	1 And-Elim
5	R	4,2 Modus Ponens
6	Q	1 And-Elim
7	Q ∧ R	5,6 And-Intro
8	S	7,3 Modus Ponens

6.034 - Spring 03 • 30

Slide 9.3.30
Finally, from lines 7 and 3, we can use modus ponens to get S. Whew! We did it!

Slide 9.3.31
The process of formal proof seems pretty mechanical. So why can't computers do it?

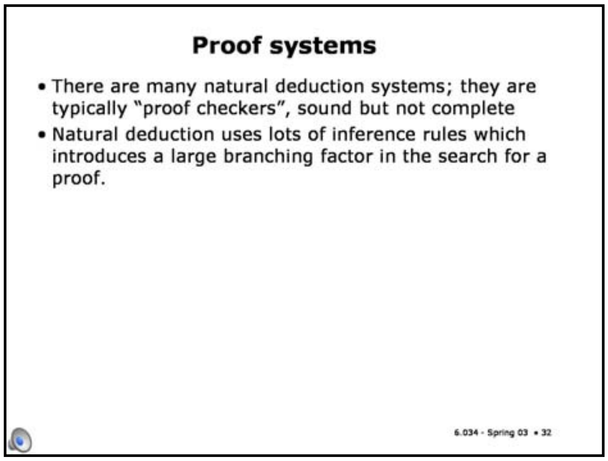
They can. For natural deduction systems, there are a lot of "proof checkers", in which you tell the system what conclusion it should try to draw from what premises. They're always sound, but nowhere near complete. You typically have to ask them to do the proof in baby steps, if you're trying to prove anything at all interesting.



Proof systems

- There are many natural deduction systems; they are typically “proof checkers”, sound but not complete

6.034 - Spring 03 • 31



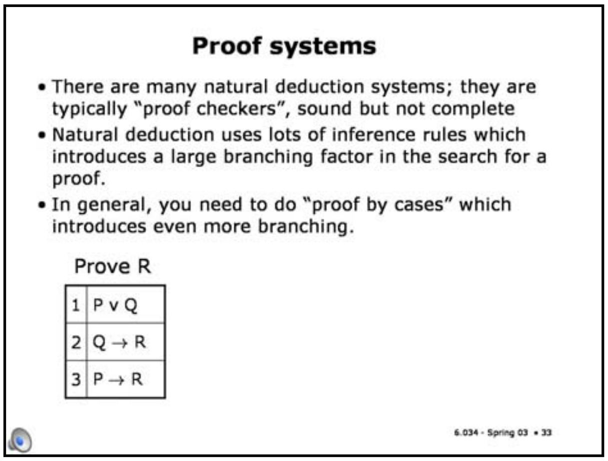
Proof systems

- There are many natural deduction systems; they are typically “proof checkers”, sound but not complete
- Natural deduction uses lots of inference rules which introduces a large branching factor in the search for a proof.

6.034 - Spring 03 • 32

Slide 9.3.32
Part of the problem is that they have a lot of inference rules, which introduces a very big branching factor in the search for proofs.

Slide 9.3.33
Another big problem is the need to do "proof by cases". What if you wanted to prove R from (P or Q), (Q implies R), and (P implies R)? You have to do it by first assuming that P is true and proving R, then assuming Q is true and proving R. And then finally applying a rule that allows you to conclude that R follows no matter what. This kind of proof by cases introduces another large amount of branching in the space.



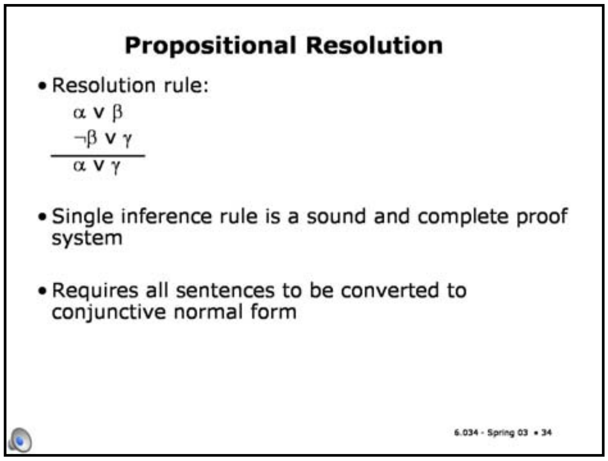
Proof systems

- There are many natural deduction systems; they are typically "proof checkers", sound but not complete
- Natural deduction uses lots of inference rules which introduces a large branching factor in the search for a proof.
- In general, you need to do "proof by cases" which introduces even more branching.

Prove R

1	$P \vee Q$
2	$Q \rightarrow R$
3	$P \rightarrow R$

6.034 - Spring 03 • 33



Propositional Resolution

- Resolution rule:

$$\frac{\alpha \vee \beta \qquad \neg\beta \vee \gamma}{\alpha \vee \gamma}$$

- Single inference rule is a sound and complete proof system
- Requires all sentences to be converted to conjunctive normal form

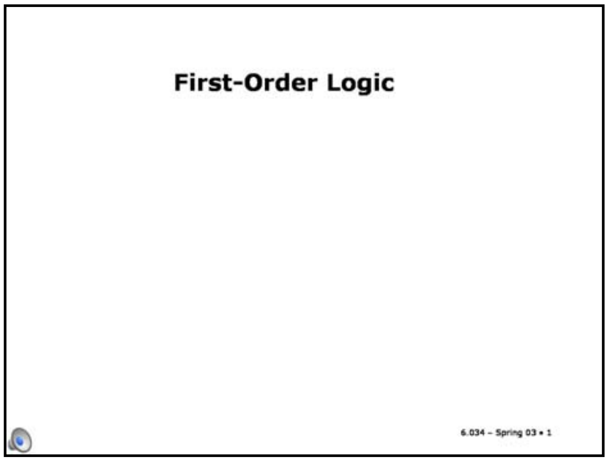
6.034 - Spring 03 • 34

Slide 9.3.34
An alternative is resolution, a single inference rule that is sound and complete, all by itself. It's not very intuitive for humans to use, but it's great for computers.

Resolution requires all sentences to be first written in a special form. So the next section will investigate that special form, and then we'll return to resolution.

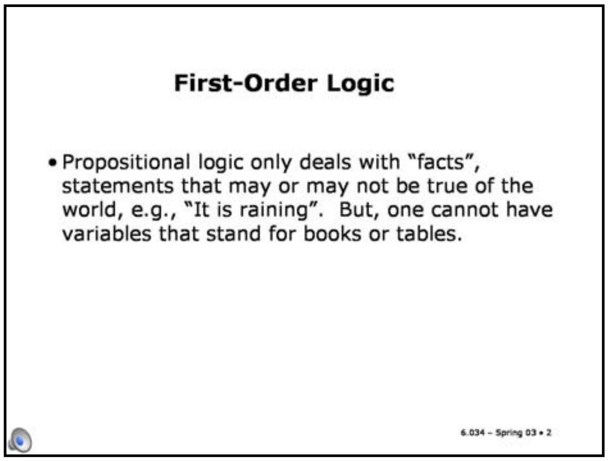
6.034 Notes: Section 9.4

Slide 9.4.1
Now we're going to start talking about first-order logic, which extends propositional logic so that we can talk about *things*.



First-Order Logic

6.034 – Spring 03 • 1



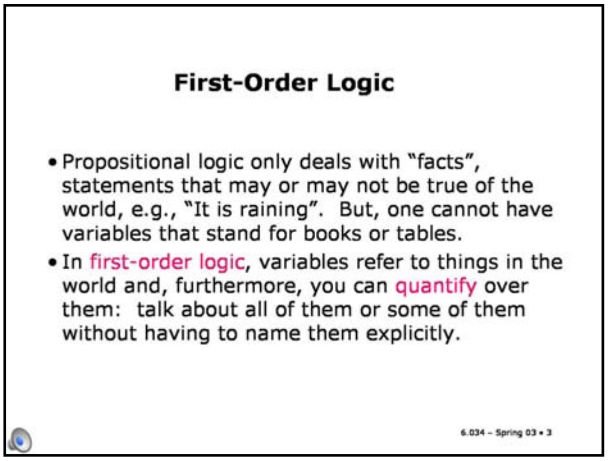
First-Order Logic

- Propositional logic only deals with "facts", statements that may or may not be true of the world, e.g., "It is raining". But, one cannot have variables that stand for books or tables.

6.034 – Spring 03 • 2

Slide 9.4.2
In propositional logic, all we had were variables that stood, not for things in the world or even quantities, but just facts, Boolean statements that might or might not be true about the world, like whether it's raining, or greater than 67 degrees; but you couldn't have variables that stood for tables or books, or the temperature, or anything like that. And as it turns out, that's an enormously limiting kind of representation.

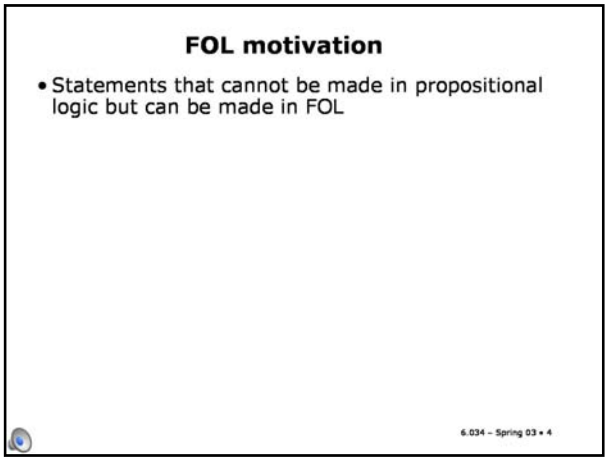
Slide 9.4.3
In first-order logic, variables refer to things in the world and you can quantify over them. That is, you can talk about all or some of them without having to name them explicitly.



First-Order Logic

- Propositional logic only deals with "facts", statements that may or may not be true of the world, e.g., "It is raining". But, one cannot have variables that stand for books or tables.
- In first-order logic, variables refer to things in the world and, furthermore, you can quantify over them: talk about all of them or some of them without having to name them explicitly.

6.034 – Spring 03 • 3



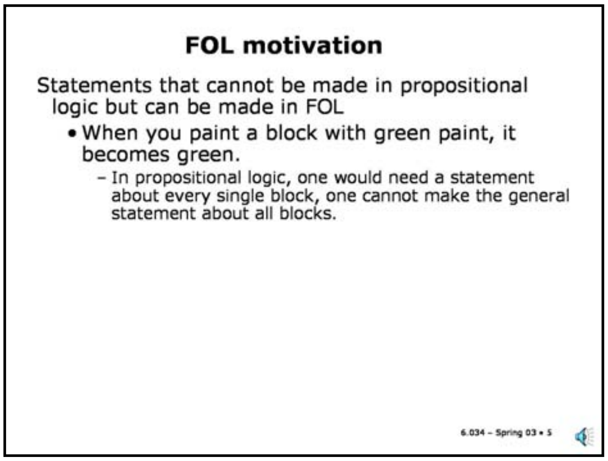
FOL motivation

- Statements that cannot be made in propositional logic but can be made in FOL

6.034 – Spring 03 • 4

Slide 9.4.4
There are lots of examples that show how propositional logic is inadequate to characterize even moderately complex domains. Here are some more examples of the kinds of things that you can say in first-order logic, but not in propositional logic.

Slide 9.4.5
"When you paint the block, it becomes green." You might have a proposition for every single aspect of the situation, like "if this block is black and I paint it, it becomes green" and "if that block is red and I paint it, it becomes green" and "if block #5 is green and I paint it, it becomes green". But you'd have to have one of those propositions for every single initial block color, or every single block, or every single object (if you have non-blocks, too) in the world. You couldn't say that, as a general fact, "after you paint something if becomes green."

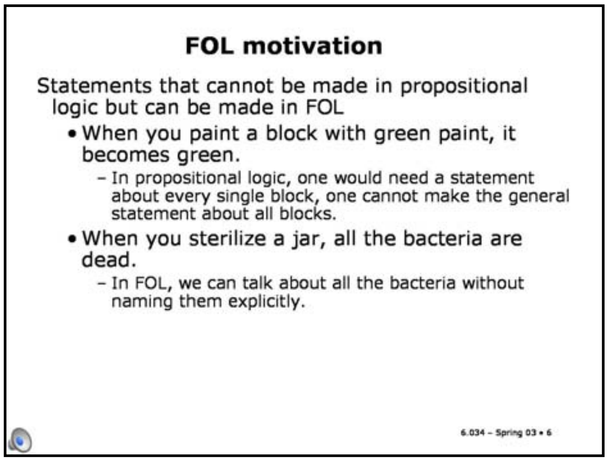


FOL motivation

Statements that cannot be made in propositional logic but can be made in FOL

- When you paint a block with green paint, it becomes green.
  - – In propositional logic, one would need a statement about every single block, one cannot make the general statement about all blocks.

6.034 – Spring 03 • 5



FOL motivation

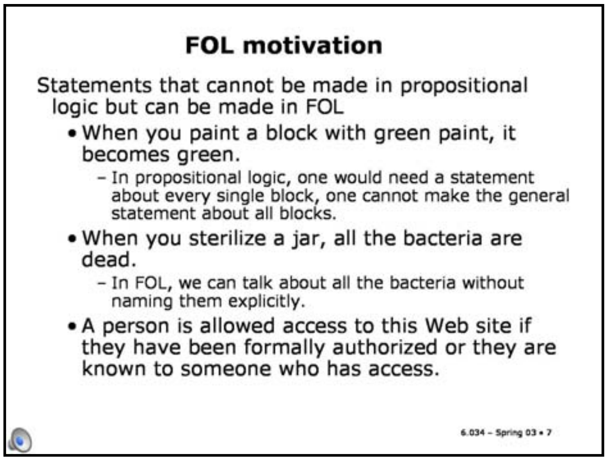
Statements that cannot be made in propositional logic but can be made in FOL

- When you paint a block with green paint, it becomes green.
  - In propositional logic, one would need a statement about every single block, one cannot make the general statement about all blocks.
- When you sterilize a jar, all the bacteria are dead.
  - In FOL, we can talk about all the bacteria without naming them explicitly.

6.034 – Spring 03 • 6

Slide 9.4.6
Let's say you want to talk about what happens when you sterilize a jar. It kills all the bacteria in the jar. Now, you don't want to have to name all the bacteria; to have to say, bacterium 57 is dead, and bacterium 93 is dead. Each one of those guys is dead. All the bacteria are dead now. So you'd like to have a way not only to talk about things in the world, but to talk about all of them, or some of them, without naming any of them explicitly.

Slide 9.4.7
In the context of providing flexible computer security, you might want to prove or try to understand whether someone should be allowed access to a web site. And you could say: a person should have access to this web site if they've been personally, formally authorized to use this web site or if they are known to someone who has access to the web site. So you could write a general rule that says that and then some other system or this system could try to prove that you should have access to the web site. In this case, what that would mean would be going to look for a chain of people that are authorized or known to one another that bottoms out in somebody who's known to this web site.

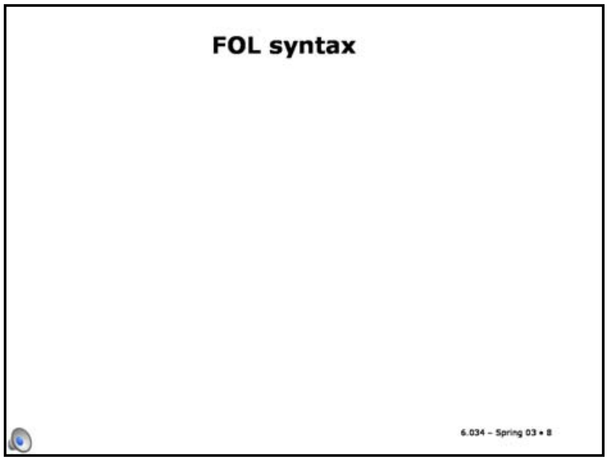


FOL motivation

Statements that cannot be made in propositional logic but can be made in FOL

- When you paint a block with green paint, it becomes green.
  - In propositional logic, one would need a statement about every single block, one cannot make the general statement about all blocks.
- When you sterilize a jar, all the bacteria are dead.
  - In FOL, we can talk about all the bacteria without naming them explicitly.
- A person is allowed access to this Web site if they have been formally authorized or they are known to someone who has access.

6.034 – Spring 03 • 7

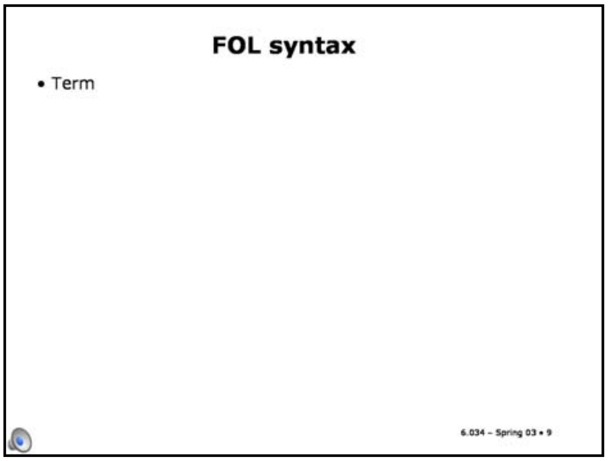


FOL syntax

6.034 – Spring 03 • 8

Slide 9.4.8
First-order logic lets us talk about things in the world. It's a logic like propositional logic, but somewhat richer and more complex. We'll go through the material in the same way that we did propositional logic: we'll start with syntax and semantics, and then do some practice with writing down statements in first-order logic.

Slide 9.4.9
The big difference between propositional logic and first-order logic is that we can talk about things, and so there's a new kind of syntactic element called a *term*. And the term, as we'll see when we do the semantics, is a name for a thing. It's an expression that somehow names a thing in the world. There are three kinds of terms:

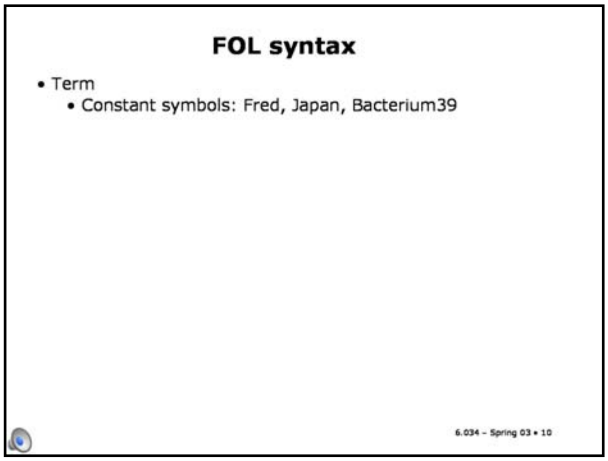


FOL syntax

- Term

6.034 – Spring 03 • 9

Slide 9.4.10
There are constant symbols. They are names like **Fred** or **Japan** or **Bacterium39**. Those are symbols that, in the context of an interpretation, name a particular thing.

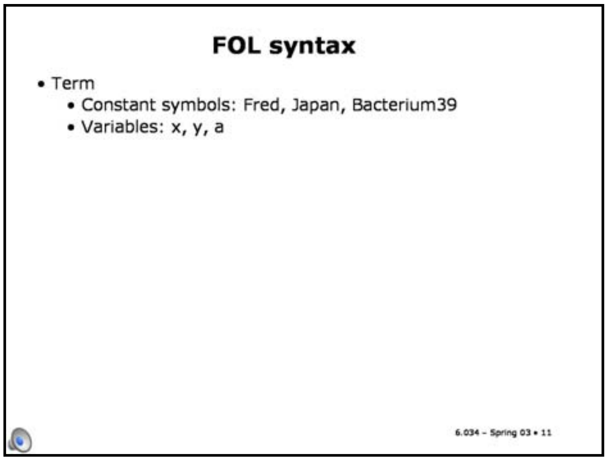


FOL syntax

- Term
  - Constant symbols: Fred, Japan, Bacterium39

6.034 – Spring 03 • 10

Slide 9.4.11
Then there are variables, which are not really syntactically differentiated from constant symbols. We'll use capital letters to start constant symbols (think of them as proper names), and lower-case letters for term variables. (It's important to note, though, that this convention is not standard, and in some logic contexts, such as the programming language Prolog, they adopt the exact opposite convention).



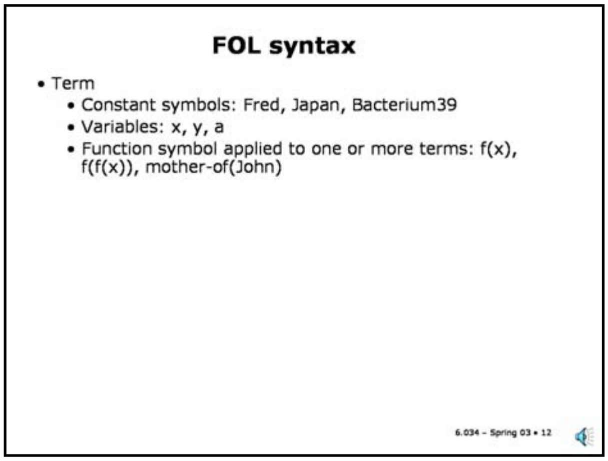
FOL syntax

- Term
  - Constant symbols: Fred, Japan, Bacterium39
  - Variables: x, y, a

6.034 – Spring 03 • 11

Slide 9.4.12
The last kind of term is a function symbol, applied to one or more terms. We'll use lower-case for function symbols as well. So another way to make a name for something is to say something like **f(x)**. If **f** is a function, you can give it a term and then **f(x)** names something. So, you might have **mother-of (John)** or **f(f(x))**. Note that a function with no terms would be a constant.

These three kinds of terms are our ways to name things in the world.



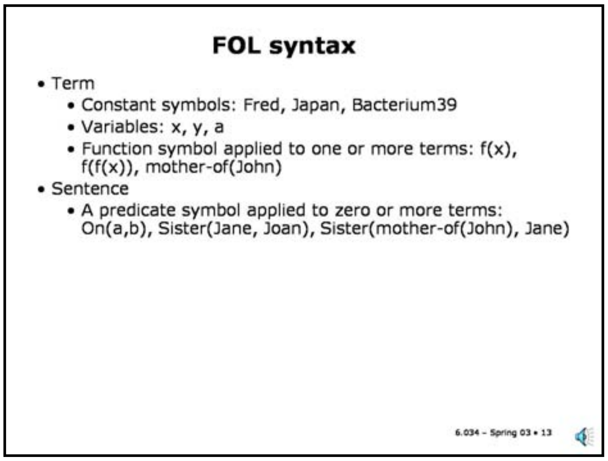
FOL syntax

- Term
  - Constant symbols: Fred, Japan, Bacterium39
  - Variables: x, y, a
  - Function symbol applied to one or more terms: f(x), f(f(x)), mother-of(John)

6.034 – Spring 03 • 12

Slide 9.4.13
In propositional logic we had sentences. Now, in first-order logic it's a little bit more complicated, but not a lot. So what's a sentence? There's another kind of symbol called a predicate symbol. A predicate symbol is applied to zero or more terms. Predicate symbols stand for relations, so we might have things like **On(A,B)** or **Sister(Jane, Joan)**. **On** and **Sister** are predicate symbols; **a**, **b**, **Jane**, **Joan**, and **mother-of(John)** are terms.

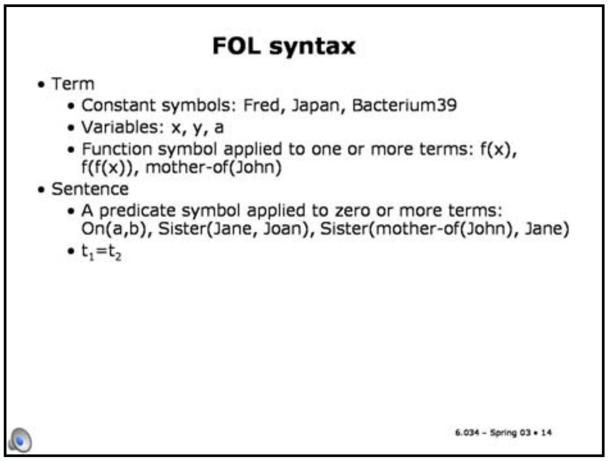
A predicate applied to zero terms is what's sometimes called a sentential variable or a propositional variable. It was our old kind of variable that we had before in propositional logic, like "it's-raining." It's a little bit of an artifice, but we'll take predicates with no arguments to be variables that have values true or false.



FOL syntax

- Term
  - Constant symbols: Fred, Japan, Bacterium39
  - Variables: x, y, a
  - Function symbol applied to one or more terms: f(x), f(f(x)), mother-of(John)
- Sentence
  - A predicate symbol applied to zero or more terms: On(a,b), Sister(Jane, Joan), Sister(mother-of(John), Jane)

6.034 – Spring 03 • 13



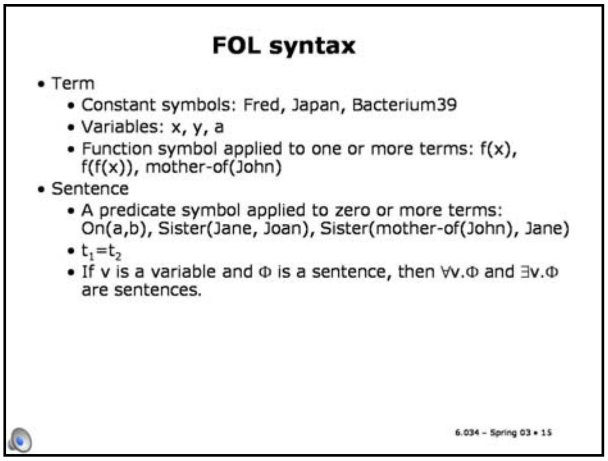
FOL syntax

- Term
  - Constant symbols: Fred, Japan, Bacterium39
  - Variables: x, y, a
  - Function symbol applied to one or more terms: f(x), f(f(x)), mother-of(John)
- Sentence
  - A predicate symbol applied to zero or more terms: On(a,b), Sister(Jane, Joan), Sister(mother-of(John), Jane)
  - $t_1=t_2$

6.034 – Spring 03 • 14

Slide 9.4.14
A sentence can also be of the form $\mathbf{t_1 = t_2}$. We're going to have one special predicate called equality. You can say this thing equals that thing, written term, equal-sign, term.

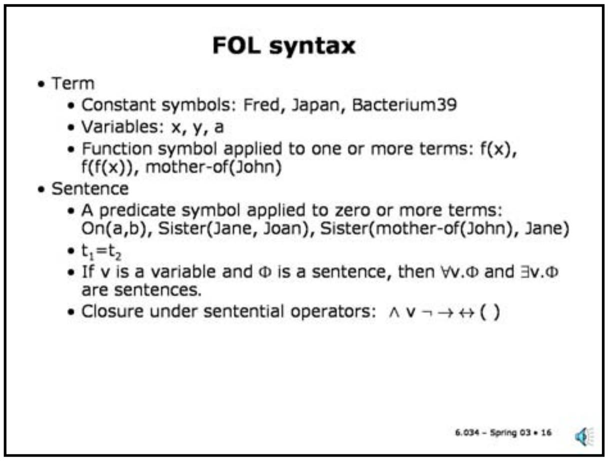
Slide 9.4.15
There are two more new constructs. If **v** is a variable and **Phi** is a sentence then **(upside-down-A v . phi)**, and **(backwards-E v . phi)** are sentences. You've probably seen these symbols before informally as "for all" and "there exists", and that's what they're going to mean for us, too.



FOL syntax

- Term
  - Constant symbols: Fred, Japan, Bacterium39
  - Variables: x, y, a
  - Function symbol applied to one or more terms: f(x), f(f(x)), mother-of(John)
- Sentence
  - A predicate symbol applied to zero or more terms: On(a,b), Sister(Jane, Joan), Sister(mother-of(John), Jane)
  - $t_1=t_2$
  - If v is a variable and Φ is a sentence, then $\forall v.\Phi$ and $\exists v.\Phi$ are sentences.

6.034 – Spring 03 • 15



FOL syntax

- Term
  - Constant symbols: Fred, Japan, Bacterium39
  - Variables: x, y, a
  - Function symbol applied to one or more terms: f(x), f(f(x)), mother-of(John)
- Sentence
  - A predicate symbol applied to zero or more terms: On(a,b), Sister(Jane, Joan), Sister(mother-of(John), Jane)
  - $t_1=t_2$
  - If v is a variable and Φ is a sentence, then $\forall v.\Phi$ and $\exists v.\Phi$ are sentences.
  - Closure under sentential operators: $\wedge \vee \neg \rightarrow \leftrightarrow (\ )$

6.034 – Spring 03 • 16

Slide 9.4.16
Finally we have closure under the sentential operators that we had before, so you can make complex sentences out of other sentences using and, or, not, implies, equivalence (also called biconditional), and parentheses, just as before in propositional logic. All that basic connective structure is still the same, but the things that we can say on either side have gotten a little bit more complicated.

All right, that's our syntax. That's what we get to write down on our page.