

Session 3: Predicate Logic and Models

What Makes a Sentence True

James Pustejovsky

Brandeis University
Department of Computer Science

CS 135: Computational Semantics

September 4, 2026

Where We Were

Session 1. Meanings are functions; composition is β -reduction.

Session 2. Every expression has a type; types license composition and determine its direction.
We derived

$$\llbracket \text{Alice likes Bob} \rrbracket = \text{like}(a, b)$$

But we have been sloppy about two things.

- What *is* $\text{like}(a, b)$? We have been treating it as a formula, without saying what formulas are.
- What makes it *true*? We sketched models in passing, but never defined satisfaction.

Today we make both precise. The output of a semantic derivation is a formula of **predicate logic**, and truth is defined by a **recursive satisfaction relation** over a model.

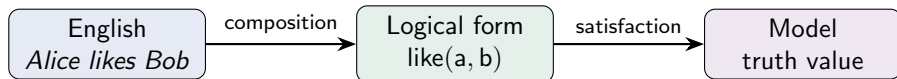
Today

- 1 The two-level architecture: why an intermediate logical language
- 2 Syntax of first-order logic
- 3 Free and bound variables; scope
- 4 Models, and assignment functions
- 5 **Satisfaction** – the recursive definition
- 6 Truth, validity, entailment
- 7 Quantifier interaction – and a preview of Session 5
- 8 Model checking as computation
- 9 A model in Haskell – and Problem Set 1

Problem Set 1 goes out at the end of today.

Why an Intermediate Language?

We could map English directly to model-theoretic objects. Montague largely did.
In practice almost everyone goes through a **logical form** instead:



Three reasons this pays.

- **Modularity.** Composition and evaluation are separate problems, debugged separately.
- **Inference.** A logical form supports entailment reasoning without ever touching a model.
- **Computability.** Over a finite model, satisfaction is a program you can run.

Problem Set 1 has exactly this shape: your denotations build formulas; the harness evaluates them.

Vocabulary

A first-order language is fixed by its **non-logical vocabulary**:

Category	Notation	Example
Individual constants	$a, b, c, \dots$	a for <i>Alice</i>
Variables	$x, y, z, \dots$	
Predicate symbols (n -ary)	$P, Q, R, \dots$	student (1-ary), like (2-ary)

And its **logical vocabulary**, which is the same for every first-order language:

$\neg \quad \wedge \quad \vee \quad \rightarrow \quad \leftrightarrow \quad \forall \quad \exists \quad (\quad)$

We omit function symbols. They are standard, and nothing in this course needs them.

Terms and Atomic Formulas

Terms

- ① Every individual constant is a term.
- ② Every variable is a term.
- ③ Nothing else is a term.

Terms are the expressions that *denote individuals* – the object-language counterpart of type e .

Atomic formulas

If P is an n -ary predicate symbol and $t_1, \dots, t_n$ are terms, then $P(t_1, \dots, t_n)$ is an atomic formula.

Examples. $\text{student}(a)$ $\text{like}(x, b)$ $\text{sleep}(y)$

Note the correspondence: an n -ary predicate symbol is the object-language version of a meaning of type $\langle e, \langle e, \dots t \rangle \rangle$ with n arrows.

Well-Formed Formulas

The recursive definition

- 1 Every atomic formula is a formula.
- 2 If φ is a formula, so is $\neg\varphi$.
- 3 If φ and ψ are formulas, so are $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$, $(\varphi \leftrightarrow \psi)$.
- 4 If φ is a formula and x a variable, then $\forall x \varphi$ and $\exists x \varphi$ are formulas.
- 5 Nothing else is a formula.

The third such definition this week – lambda terms, semantic types, and now formulas. [Base case, constructors, closure clause](#). It is worth noticing the pattern, because every proof about these objects is by induction on the same structure.

Examples.

$\neg \text{sleep}(\text{a})$ $\forall x [\text{student}(x) \rightarrow \text{sleep}(x)]$ $\exists x \forall y \text{like}(y, x)$

Scope of a Quantifier

In $\forall x \varphi$, the formula φ is the **scope** of the quantifier.

$$\underbrace{\forall x \left[\text{student}(x) \rightarrow \underbrace{\exists y \left[\text{book}(y) \wedge \text{read}(x, y) \right]}_{\text{scope of } \exists y} \right]}_{\text{scope of } \forall x}$$

Brackets matter. Compare:

$\forall x [\text{student}(x) \rightarrow \text{sleep}(x)]$ every student sleeps

$\forall x \text{ student}(x) \rightarrow \text{sleep}(x)$ if everything is a student, then *it* sleeps – *x* is free!

The second is not a sentence at all. We will always bracket explicitly.

Free and Bound, Again

An occurrence of x is **bound** if it lies in the scope of a quantifier $\forall x$ or $\exists x$; otherwise it is **free**.

Formula	Free variables	Sentence?
sleep(a)	none	yes
sleep(x)	x	no
$\forall x$ sleep(x)	none	yes
$\forall x$ like(x, y)	y	no
$\exists y \forall x$ like(x, y)	none	yes

A formula with no free variables is a **sentence** (or *closed formula*).

Only sentences have truth values. A formula with a free variable is true *relative to an assignment* – which is why the next section introduces assignment functions properly.

Two Notions of Binding, Side by Side

You have now seen binding twice this week. They are related but not identical.

	λ -binding (Session 1)	Quantifier binding (today)
Form	$\lambda x.\varphi$	$\forall x \varphi, \exists x \varphi$
Creates	a function	a truth value
Discharged by	application (β)	satisfaction over the domain
α -equivalent?	yes	yes

Both bind variables; both are subject to capture; both are α -equivalent under renaming.

And both are things that Theme 5 has to reconstruct. The wedge product binds in a third sense entirely – role to filler – and whether it can do the work of *either* of these is one of the open questions from Session 1.

Models, Precisely

A model $\mathcal{M} = \langle \mathcal{D}, I \rangle$

- $\mathcal{D}$ – a non-empty set, the **domain**
- I – the **interpretation function**, with
 - $I(c) \in \mathcal{D}$ for each individual constant c
 - $I(P) \subseteq \mathcal{D}^n$ for each n -ary predicate symbol P

$I(P)$ is the **extension** of P : the set of things (or tuples of things) it is true of.

- For a 1-ary predicate: a subset of $\mathcal{D}$.
- For a 2-ary predicate: a set of ordered pairs.

Recall from Session 2: a subset of $\mathcal{D}$ and a function $\mathcal{D} \rightarrow \{0, 1\}$ are the same object. We move between the two freely – sets are easier to write, functions are easier to compose.

Our Running Model

$$\mathcal{D} = \{\text{Alice, Bob, Carol, Fido}\}$$

Symbol	Interpretation
a, b, c	Alice, Bob, Carol
student	{Alice, Bob, Carol}
dog	{Fido}
sleep	{Carol, Fido}
like	{(Alice, Bob), (Alice, Fido), (Bob, Fido)}

The same model as Session 2, and essentially the one in Problem Set 1.

Convention: like contains (x, y) when x likes y . So Alice likes Bob, and nobody likes Alice.

Assignment Functions

A model interprets *constants*. It says nothing about *variables*.

Assignment

An assignment g is a function from variables to elements of $\mathcal{D}$.

For $d \in \mathcal{D}$, write $g[x \mapsto d]$ for the assignment that is like g everywhere except that it sends x to d .

Denotation of terms, relative to $\mathcal{M}$ and g :

$$\llbracket c \rrbracket^{\mathcal{M},g} = I(c) \qquad \llbracket x \rrbracket^{\mathcal{M},g} = g(x)$$

The modified assignment $g[x \mapsto d]$ is the entire technical content of quantification. Everything else follows from it.

Satisfaction: The Definition

$\mathcal{M}, g \models \varphi$ (“ φ is satisfied in $\mathcal{M}$ under g ”)

$$\mathcal{M}, g \models P(t_1, \dots, t_n) \iff ([t_1]^{\mathcal{M}, g}, \dots, [t_n]^{\mathcal{M}, g}) \in I(P)$$

$$\mathcal{M}, g \models \neg\varphi \iff \mathcal{M}, g \not\models \varphi$$

$$\mathcal{M}, g \models (\varphi \wedge \psi) \iff \mathcal{M}, g \models \varphi \text{ and } \mathcal{M}, g \models \psi$$

$$\mathcal{M}, g \models (\varphi \vee \psi) \iff \mathcal{M}, g \models \varphi \text{ or } \mathcal{M}, g \models \psi$$

$$\mathcal{M}, g \models (\varphi \rightarrow \psi) \iff \mathcal{M}, g \not\models \varphi \text{ or } \mathcal{M}, g \models \psi$$

$$\mathcal{M}, g \models \forall x \varphi \iff \text{for every } d \in \mathcal{D} : \mathcal{M}, g[x \mapsto d] \models \varphi$$

$$\mathcal{M}, g \models \exists x \varphi \iff \text{for some } d \in \mathcal{D} : \mathcal{M}, g[x \mapsto d] \models \varphi$$

The last two clauses are the heart of it. Note the recursion: satisfaction of a quantified formula is defined in terms of satisfaction of a *simpler* formula, under *modified* assignments.

Truth in a Model

Satisfaction is relative to an assignment. Truth is not.

Truth

For a **sentence** φ (no free variables):

$$\mathcal{M} \models \varphi \quad \text{iff} \quad \mathcal{M}, g \models \varphi \text{ for every assignment } g$$

Why this is well-behaved. If φ has no free variables, then satisfaction does not depend on g at all – one can prove by induction that changing g at a variable not free in φ never changes the outcome.

So for sentences: *every* assignment works, or none does.

This is the point of insisting that only sentences have truth values, and the reason our lexical entries are always closed terms.

Worked Evaluation I

Evaluate $\forall x [\text{student}(x) \rightarrow \text{sleep}(x)]$ in our model.

By the $\forall$ clause, check every $d \in \mathcal{D}$:

d	$\text{student}(d)$	$\text{sleep}(d)$	$\text{student}(d) \rightarrow \text{sleep}(d)$
Alice	1	0	0
Bob	1	0	0
Carol	1	1	1
Fido	0	1	1

Not every row is 1, so $\mathcal{M} \not\models \forall x [\text{student}(x) \rightarrow \text{sleep}(x)]$ – *Every student sleeps is false.*

Note the last row: Fido satisfies the conditional *vacuously*, because he is not a student. This is the standard behaviour of material implication, and it is why $\forall$ pairs with $\rightarrow$ while $\exists$ pairs with $\wedge$.

Worked Evaluation II

Evaluate $\exists x [\text{dog}(x) \wedge \text{sleep}(x)]$.

By the $\exists$ clause, we need *some* d :

d	$\text{dog}(d)$	$\text{sleep}(d)$	$\text{dog}(d) \wedge \text{sleep}(d)$
Alice	0	0	0
Bob	0	0	0
Carol	0	1	0
Fido	1	1	1

Fido works, so $\mathcal{M} \models \exists x [\text{dog}(x) \wedge \text{sleep}(x)]$ – *Some dog sleeps is true*.

Why $\wedge$ and not $\rightarrow$? With $\rightarrow$, the formula $\exists x [\text{dog}(x) \rightarrow \text{sleep}(x)]$ would be true whenever *anything* fails to be a dog – satisfied by Alice, vacuously. That is not what *some dog sleeps* means.

Practice

Evaluate in the running model. Two minutes, with a neighbour.

- 1 $\text{like}(a, b)$
- 2 $\neg \exists x \text{ like}(x, a)$
- 3 $\forall x [\text{dog}(x) \rightarrow \text{sleep}(x)]$
- 4 $\exists x [\text{student}(x) \wedge \neg \text{sleep}(x)]$

Practice

Evaluate in the running model. Two minutes, with a neighbour.

- 1 $\text{like}(a, b)$
- 2 $\neg \exists x \text{ like}(x, a)$
- 3 $\forall x [\text{dog}(x) \rightarrow \text{sleep}(x)]$
- 4 $\exists x [\text{student}(x) \wedge \neg \text{sleep}(x)]$

Answers.

- 1 **True** – $(\text{Alice}, \text{Bob}) \in I(\text{like})$.
- 2 **True** – no pair in $I(\text{like})$ has Alice in second position.
- 3 **True** – Fido is the only dog, and he sleeps.
- 4 **True** – Alice is a student and does not sleep.

Notice that (4) is the negation of Worked Evaluation I, and gets the opposite value. That is not an accident – see the next section.

Three Logical Notions

Truth is relative to a model. The interesting notions quantify *over* models.

Notion	Notation	Holds when
Satisfiable	—	<i>some</i> model makes φ true
Valid	$\models \varphi$	<i>every</i> model makes φ true
Entailment	$\Gamma \models \varphi$	every model making all of Γ true makes φ true

Entailment is the payoff. It is the formal counterpart of *it follows that*, and what a semantic theory is ultimately accountable to:

Every student sleeps, Alice is a student $\Rightarrow$ Alice sleeps

No competent speaker of English needs to be told this. A semantic theory that cannot derive it is not a theory of meaning.

Entailment, Verified

Show: $\{ \forall x[\text{student}(x) \rightarrow \text{sleep}(x)], \text{student}(a) \} \models \text{sleep}(a)$

Proof sketch. Let $\mathcal{M}$ be *any* model satisfying both premises.

- 1 From the first premise and the $\forall$ clause: for every $d \in \mathcal{D}$, if $d \in I(\text{student})$ then $d \in I(\text{sleep})$.
- 2 Instantiate at $d = I(a)$.
- 3 From the second premise: $I(a) \in I(\text{student})$.
- 4 Therefore $I(a) \in I(\text{sleep})$, i.e. $\mathcal{M} \models \text{sleep}(a)$. □

Note what the argument did *not* use: any particular model. The reasoning goes through for all of them at once, which is exactly what entailment demands.

This is also why the logical form is worth having. We reasoned entirely in the object language and never evaluated anything.

Logical Equivalence

φ and ψ are **logically equivalent** ($\varphi \equiv \psi$) when they are satisfied in exactly the same models under the same assignments.

Equivalence	Name
$\neg\neg\varphi \equiv \varphi$	double negation
$\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$	material implication
$\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi$	De Morgan
$\neg\forall x \varphi \equiv \exists x \neg\varphi$	quantifier duality
$\neg\exists x \varphi \equiv \forall x \neg\varphi$	quantifier duality
$\forall x \forall y \varphi \equiv \forall y \forall x \varphi$	like quantifiers commute
$\exists x \exists y \varphi \equiv \exists y \exists x \varphi$	like quantifiers commute

Conspicuously absent from this list: $\forall x \exists y \varphi$ and $\exists y \forall x \varphi$.

Unlike Quantifiers Do Not Commute

$$\forall x \exists y \text{ like}(x, y) \quad \text{versus} \quad \exists y \forall x \text{ like}(x, y)$$

- **Left:** everyone likes *something* – possibly a different thing each.
- **Right:** there is *one thing* that everyone likes.

The entailment goes one way only

$$\exists y \forall x \varphi \models \forall x \exists y \varphi \quad \text{but} \quad \forall x \exists y \varphi \not\models \exists y \forall x \varphi$$

If there is one universally-liked thing, then certainly everyone likes something. The converse fails.

A counter-model is easy to build – and worth building yourself before the next slide.

A Counter-Model

Take $\mathcal{D} = \{1, 2\}$ and $I(\text{like}) = \{(1, 1), (2, 2)\}$.



- $\forall x \exists y \text{ like}(x, y)$: 1 likes 1 ✓, 2 likes 2 ✓. **True.**
- $\exists y \forall x \text{ like}(x, y)$: is 1 liked by both? No. Is 2? No. **False.**

Quantifier **order** is semantically significant, and the two orders are not interderivable. Two formulas, same atoms, same quantifiers – different truth conditions.

Why This Matters for Natural Language

Every student read a book.

One string. Two logical forms.

Surface ($\forall > \exists$) $\forall x [\text{student}(x) \rightarrow \exists y [\text{book}(y) \wedge \text{read}(x, y)]]$

Inverse ($\exists > \forall$) $\exists y [\text{book}(y) \wedge \forall x [\text{student}(x) \rightarrow \text{read}(x, y)]]$

The problem for compositionality. The syntax gives us one structure. Composition, as we defined it in Session 2, is a function of the structure. So where does the second reading come from?

This is the central puzzle of **Session 5**, and the capstone of Theme 1. You will derive both readings in Problem Set 1 – and Part E of that assignment asks you to say which entails which.

Satisfaction Is an Algorithm

Look again at the $\forall$ clause:

$$\mathcal{M}, g \models \forall x \varphi \iff \text{for every } d \in \mathcal{D} : \mathcal{M}, g[x \mapsto d] \models \varphi$$

If $\mathcal{D}$ is **finite**, this is a for loop.

Logic	Computation
$\forall x \varphi$	check φ for every element of $\mathcal{D}$
$\exists x \varphi$	check φ until one element succeeds
$\varphi \wedge \psi$	conjunction of two Booleans
$P(t_1, \dots, t_n)$	membership test in a finite set

Over a finite model, satisfaction is decidable – and the decision procedure is a direct transcription of the definition.

Over infinite models it is not. First-order validity is undecidable (Church, Turing, 1936) – the same year as the lambda calculus, and not a coincidence.

A Model in Haskell

The definitions you just saw, as code you can run.

Why Haskell, and Why Now

Everything today has been a **definition by recursion over structure**. Haskell is a language in which such definitions are transcribed almost literally.

- Functions are values; you write them without naming them (λ becomes `\`).
- Types are checked; a semantic type mismatch becomes a compile error.
- No mutable state, so an expression's value depends only on its parts – **referential transparency is compositionality, enforced by the compiler**.

You are not being asked to learn Haskell as a language. You are being asked to read and write a few dozen lines of it as *notation for the semantics you already know*.

Labs 1–3 cover the mechanics. Session 4 covers the notation properly. Today is a first look, so that the file you receive at the end of class is not a surprise.

The Domain

```
data Entity
  = Alice | Bob | Carol | Fido
  deriving (Eq, Ord, Show, Enum, Bounded)

domain :: [Entity]
domain = [minBound .. maxBound]
```

- `data Entity = ...` declares the domain $\mathcal{D}$ by listing its members.
- `domain` is that set as a list – **this is what quantifiers will range over**.
- `deriving` asks the compiler to generate equality, ordering, and printing for free.

This is $\mathcal{D} = \{\text{Alice}, \text{Bob}, \text{Carol}, \text{Fido}\}$, in four lines.

Types, and the Interpretation Function

```
type T      = Bool                -- the type t
type ET     = Entity -> T         -- the type <e,t>
type TV     = Entity -> Entity -> T -- the type <e,<e,t>>

studentSet, dogSet, sleepSet :: [Entity]
studentSet = [Alice, Bob, Carol]
dogSet     = [Fido]
sleepSet   = [Carol, Fido]

likePairs :: [(Entity, Entity)]
likePairs = [(Alice,Bob), (Alice,Fido), (Bob,Fido)]
```

The type declarations are **the semantic types of Session 2**, spelled in Haskell. The sets below are I – the interpretation function, written out as extensions.

Compare the table three slides back. This is the same model, in a form a compiler can check.

From Sets to Predicates

```
inSet :: [Entity] -> ET
inSet xs = \x -> x 'elem' xs

inRel :: [(Entity,Entity)] -> TV
inRel ps = \obj subj -> (subj, obj) 'elem' ps

student, dog, sleeps :: ET
student = inSet studentSet
dog      = inSet dogSet
sleeps   = inSet sleepSet

likes :: TV
likes = inRel likePairs
```

`inSet` is the **characteristic function** construction from Session 2: it turns a set into a function of type $\langle e, t \rangle$.

Note `\x -> ...` – that backslash is λ . And note that `inRel` takes the *object* first, exactly as we argued in Session 2.

Quantifiers Are Loops

The satisfaction clauses, transcribed:

```
forall :: ET -> T
forall p = all p domain      -- for EVERY d in D: p(d)

exists :: ET -> T
exists p = any p domain      -- for SOME d in D: p(d)
```

Compare:

$$\mathcal{M}, g \models \forall x \varphi \iff \text{for every } d \in \mathcal{D} : \mathcal{M}, g[x \mapsto d] \models \varphi$$

The assignment update $g[x \mapsto d]$ is Haskell's function application: $p \ d$. Because p is a function of x , applying it to d is evaluating φ under the modified assignment.

This works because domain is finite. An infinite domain would make `all` diverge – the computational shadow of undecidability.

Evaluating Sentences

Now the two worked examples from earlier, as code:

```
-- Every student sleeps.  
s1 :: T  
s1 = forAll (\x -> not (student x) || sleeps x)  
  
-- Some dog sleeps.  
s2 :: T  
s2 = exists (\x -> dog x && sleeps x)
```

Logic	Haskell	Value
$\forall x[\text{student}(x) \rightarrow \text{sleep}(x)]$	s1	False
$\exists x[\text{dog}(x) \wedge \text{sleep}(x)]$	s2	True

Same answers we computed by hand – but now the computer checks them. That is what makes Problem Set 1 testable.

Note not $p \mid\mid q$ for $p \rightarrow q$: material implication, exactly as in the equivalence table.

What You Have Just Seen

Session 2–3 concept	Haskell	In PS 1
domain $\mathcal{D}$	<code>data Entity, domain</code>	given
type e, t	<code>Entity, Bool</code>	given
type $\langle e, t \rangle$	<code>Entity -> Bool</code>	given
interpretation I	the sets and pair lists	given
characteristic function	<code>inSet, inRel</code>	given
$\lambda x. \varphi$	<code>\x -> ...</code>	you write these
$\forall, \exists$	all, any over domain	you write these
composition rules	function application	you write these

Everything in the “given” rows is already written for you. The assignment is the semantics, not the infrastructure.

Problem Set 1

Out today. Due Friday 25 September.

A working Haskell file with roughly twenty holes marked `-- TODO`. You fill in:

- lexical denotations – nouns, verbs, and the determiners *every*, *a*, *no*;
- composition rules – function application and predicate modification;
- derivations for seven sentences;
- **both scope readings** of *Every student read a book*.

Plus a short writeup: the scope reasoning, one type error you hit and what it told you, and one limitation of the fragment.

Run `runghc Fragment.hs` at any point to see `TODO` / `FAIL` / `PASS` per sentence. Unfilled holes do not crash the run.

Lab this week and next covers the Haskell you need. Session 4 covers the notation properly and builds the Fragment together.

Summary

Concept	Key point
Two-level architecture	English $\rightarrow$ logical form $\rightarrow$ model
Terms	constants and variables; denote individuals
Formulas	recursive definition; atoms, connectives, quantifiers
Scope	the formula a quantifier governs; bracket explicitly
Sentence	closed formula – only these have truth values
Model $\langle \mathcal{D}, I \rangle$	domain plus extensions for each symbol
Assignment g	interprets variables; $g[x \mapsto d]$ drives quantification
Satisfaction	recursive: $\forall$ checks all d , $\exists$ checks some d
Truth	satisfaction under every assignment; well defined for sentences
Entailment	every model of the premises is a model of the conclusion
Quantifier duality	$\neg \forall x \varphi \equiv \exists x \neg \varphi$
Quantifier order	$\forall \exists \not\equiv \exists \forall$; entailment runs one way only
Finite models	satisfaction becomes a decidable computation

Coming Up

Session 4 (Tuesday): The Fragment of English

- The lexicon as a set of typed denotations
- Composition rules, stated generally
- Building and checking a fragment end to end
- **Haskell notation, properly** – enough to write, not just read

Reading: EU Chapter 6.

Then Session 5: Quantifiers and scope – where *Every student read a book* gets its two readings, and where composition stops being mechanical.

Reminder: if your linear algebra self-assessment showed a weak section – especially Section E, determinants and orientation – Weeks 2–5 are when there is room to fix it. The review guide is on the course site.