

# Compositional Semantics from an NLP Perspective

Types, Lambdas, and Categorical Grammar  
(companion to Session 2)

James Pustejovsky

Brandeis University  
Department of Computer Science

CS 135: Computational Semantics

September 1, 2026

## About This Deck

Session 2 developed semantic types the way a **formal semanticist** would: Frege, Montague, denotations, models.

This companion deck covers *the same machinery* the way a **computational linguist** would: starting from a task, building toward a system, and using the notation that dominates the NLP literature.

|                  | Session 2                | This deck                         |
|------------------|--------------------------|-----------------------------------|
| Starts from      | compositionality         | a database query task             |
| Syntax-semantics | phrase structure + rules | <b>CCG</b> categories             |
| Motivation       | what meaning <i>is</i>   | what a system must <i>compute</i> |
| Extra material   | —                        | events, semantic roles, learning  |

Same lambda calculus. Same types. Different door into the building. Use whichever one makes the ideas stick.

# Today

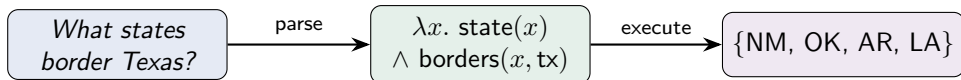
- ① A task: answering questions against a database
- ② Meaning representations, and what we want from them
- ③ Where do logical forms come from? The composition problem
- ④ Lambda calculus as glue
- ⑤ Types, and why they earn their keep
- ⑥ Combinatory Categorical Grammar
- ⑦ Events and semantic roles
- ⑧ Learning, and why any of this survives the LLM era

## Start With a Task

You have a database of US geography. A user types:

*What states border Texas?*

Your system has to turn that string into something **executable**.



The middle box is a **meaning representation**. Everything in this deck is about how to build it – systematically, for sentences nobody has seen before.

# Why Not Just Pattern-Match?

A first instinct: write templates.

```
"What states border X?" -> state(?) & borders(?, X)
"What states are in X?" -> state(?) & in(?, X)
```

This works. Briefly.

| Query                                               | Problem                         |
|-----------------------------------------------------|---------------------------------|
| <i>What states border Texas and Oklahoma?</i>       | conjunction inside the argument |
| <i>What is the largest state bordering Texas?</i>   | superlative over a filtered set |
| <i>What states border states that border Texas?</i> | recursion                       |

The last row is fatal. Language nests without bound, so no finite template list can cover it.

**We need to build meanings out of parts** – which is to say, we need compositionality, whether or not we call it that.

# What We Want From a Meaning Representation

- ① **Unambiguous.** One representation per reading. If a sentence has two readings, we should get two representations.
- ② **Executable.** We can run it against a database, knowledge base, or model and get an answer.
- ③ **Compositional.** Built from the parts, so novel sentences work.
- ④ **Inference-supporting.** If *Texas borders Oklahoma*, we should be able to derive *Some state borders Oklahoma*.
- ⑤ **Canonical.** Paraphrases should ideally converge.

|                   | Unambig. | Exec. | Comp.   | Infer.  | Canon.  |
|-------------------|----------|-------|---------|---------|---------|
| First-order logic | ✓        | ✓     | ✓       | ✓       | partial |
| SQL               | ✓        | ✓     | partial | —       | —       |
| AMR               | partial  | —     | partial | partial | ✓       |
| Raw embeddings    | —        | —     | —       | —       | —       |

The last row is Part II of this course, and its empty middle column is the whole problem.

## Other Meaning Representations You Will Meet

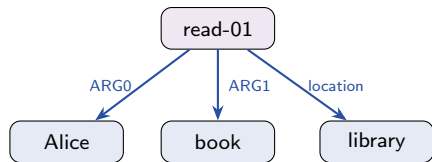
| Formalism             | Looks like                                                      | Used for                  |
|-----------------------|-----------------------------------------------------------------|---------------------------|
| FOL / lambda calculus | $\lambda x.\text{state}(x) \wedge \text{borders}(x, \text{tx})$ | QA, semantic parsing      |
| SQL                   | SELECT ... WHERE ...                                            | database interfaces       |
| SPARQL                | triple patterns                                                 | knowledge graphs          |
| Lambda-DCS            | State $\sqcap$ Border.Texas                                     | Freebase QA               |
| AMR                   | rooted labelled graph                                           | broad-coverage annotation |
| Python / API calls    | get_states(borders="TX")                                        | tool use, agents          |

They differ in expressiveness and in what they can be executed against. **They do not differ in the compositional problem:** every one of them has to be *built* from a sentence, and the building is the hard part.

Learn the lambda calculus version and the rest are notation. That is why this course spends its time there.

## An AMR, for Comparison

*Alice read the book in the library.*



The corresponding logical form:

$$\begin{aligned} &\exists e [ \text{read}(e) \wedge \text{arg0}(e, \text{alice}) \\ &\wedge \text{arg1}(e, \text{book}) \wedge \text{loc}(e, \text{library}) ] \end{aligned}$$

An AMR is a neo-Davidsonian logical form, drawn as a graph. Nodes are the event and its participants; edges are the role predicates.

AMR deliberately drops quantifier scope, tense, and articles – which makes it annotatable at scale, and unable to distinguish the two readings of *Every student read a book*. A real engineering tradeoff, and worth knowing about.

# First-Order Logic as the Target

| Sentence                           | Logical form                                                                                    |
|------------------------------------|-------------------------------------------------------------------------------------------------|
| <i>Texas borders Oklahoma</i>      | $\text{borders}(\text{tx}, \text{ok})$                                                          |
| <i>Texas is a state</i>            | $\text{state}(\text{tx})$                                                                       |
| <i>What states border Texas?</i>   | $\lambda x. \text{state}(x) \wedge \text{borders}(x, \text{tx})$                                |
| <i>Every state borders a state</i> | $\forall x[\text{state}(x) \rightarrow \exists y[\text{state}(y) \wedge \text{borders}(x, y)]]$ |
| <i>No state borders Hawaii</i>     | $\neg \exists x[\text{state}(x) \wedge \text{borders}(x, \text{hi})]$                           |

- **Constants** (tx, ok) name entities in the database.
- **Predicates** (state, borders) name tables or columns.
- **Variables** ( $x$ ,  $y$ ) are what the query returns or quantifies over.

Note row three: a question is a *one-place predicate*. Executing it means finding everything that satisfies it – exactly a `SELECT ... WHERE`.

# The Database Is the Model

Session 3 will define a **model** as a domain plus an interpretation function. In this setting, that is just your data.

| Logic                | Database                    | Example                       |
|----------------------|-----------------------------|-------------------------------|
| domain $\mathcal{D}$ | the set of rows/entities    | all 50 states, cities, rivers |
| constant             | a primary key               | $tx \mapsto \text{row Texas}$ |
| 1-place predicate    | a table, or a column filter | $\text{state}(x)$             |
| 2-place predicate    | a join table                | $\text{borders}(x,y)$         |
| truth value          | query returns rows or not   |                               |

**Model-theoretic semantics is not an abstraction here** – it is the specification of what your query engine does. “Truth in a model” is “the row is present”.

Historically this is GeoQuery (Zelle & Mooney 1996) and ATIS – the datasets that made semantic parsing a subfield.

# The Problem: Where Do Logical Forms Come From?

We know what we want:

*Texas borders Oklahoma*  $\longrightarrow$  *borders(tx, ok)*

We know the words contribute pieces:

---

|                 |                 |
|-----------------|-----------------|
| <i>Texas</i>    | tx              |
| <i>Oklahoma</i> | ok              |
| <i>borders</i>  | borders(__, __) |

---

**But how does a piece with holes in it combine with a piece that fills a hole?**

We need notation for “a formula with a labelled gap”. That is exactly what lambda abstraction is.

# Lambda Notation in Sixty Seconds

$\lambda x. \varphi$  means “the function that takes  $x$  and returns  $\varphi$ ”

**Application** substitutes the argument for the variable:

$$(\lambda x. \text{state}(x))(\text{tx}) \rightsquigarrow \text{state}(\text{tx})$$

**Two arguments** means two nested lambdas:

$$\lambda y. \lambda x. \text{borders}(x, y)$$

and you apply them one at a time:

$$(\lambda y. \lambda x. \text{borders}(x, y))(\text{ok}) \rightsquigarrow \lambda x. \text{borders}(x, \text{ok})$$

That intermediate result –  $\lambda x. \text{borders}(x, \text{ok})$  – is the meaning of the verb phrase *borders Oklahoma*. It is a formula with one gap left.

## Partial Application Is Phrase Structure

| Expression                    | Lambda term                                   | Gaps left |
|-------------------------------|-----------------------------------------------|-----------|
| <i>borders</i>                | $\lambda y. \lambda x. \text{ borders}(x, y)$ | 2         |
| <i>borders Oklahoma</i>       | $\lambda x. \text{ borders}(x, \text{ok})$    | 1         |
| <i>Texas borders Oklahoma</i> | $\text{ borders}(\text{tx}, \text{ok})$       | 0         |

Every application fills one gap, and each intermediate result corresponds to a **real constituent** of the sentence.

The lambda calculus was not designed for language, but partial application lines up exactly with phrase structure – which is why it became the standard glue for semantic parsing.

Note the object comes first in the lexical entry, so that applying it once gives you the VP.

# A Full Derivation

*Texas borders Oklahoma.*

**Lexicon.**

*Texas*  $\rightsquigarrow$  tx

*Oklahoma*  $\rightsquigarrow$  ok

*borders*  $\rightsquigarrow$   $\lambda y. \lambda x. \text{borders}(x, y)$

**Step 1 – verb + object.**

$(\lambda y. \lambda x. \text{borders}(x, y))(ok) \rightsquigarrow \lambda x. \text{borders}(x, ok)$

**Step 2 – VP + subject.**

$(\lambda x. \text{borders}(x, ok))(tx) \rightsquigarrow \text{borders}(tx, ok)$

**Executable.** Two reductions. And *Oklahoma borders Texas* gives  $\text{borders}(ok, tx)$  – a different query, as it should be.

## A Question, Derived

*What states border Texas?*

*states*  $\rightsquigarrow \lambda x. \text{state}(x)$

*border Texas*  $\rightsquigarrow \lambda x. \text{borders}(x, \text{tx})$

*what*  $\rightsquigarrow$  conjoin two one-place predicates

$\lambda x. \text{state}(x) \wedge \text{borders}(x, \text{tx})$

**Executing it** means: iterate over the domain, keep the  $x$  that satisfy the body.

```
SELECT s.name FROM states s
JOIN borders b ON s.id = b.state1
WHERE b.state2 = 'Texas';
```

The logical form and the SQL are the same query. One is compositional and one is not – which is why we generate the first and compile to the second.

## Practice: Build the Logical Form

Give a lambda term for each. Assume predicates *state*, *city*, *river*, *borders*, *in*, *large*.

- ① *Oklahoma borders Texas*
- ② *large states*
- ③ *cities in Texas*
- ④ *states bordering Texas*
- ⑤ *large cities in states bordering Texas*

## Practice: Build the Logical Form

Give a lambda term for each. Assume predicates *state*, *city*, *river*, *borders*, *in*, *large*.

- 1 *Oklahoma borders Texas*
- 2 *large states*
- 3 *cities in Texas*
- 4 *states bordering Texas*
- 5 *large cities in states bordering Texas*

### Answers.

- 1  $\text{borders}(\text{ok}, \text{tx})$
- 2  $\lambda x. \text{state}(x) \wedge \text{large}(x)$
- 3  $\lambda x. \text{city}(x) \wedge \text{in}(x, \text{tx})$
- 4  $\lambda x. \text{state}(x) \wedge \text{borders}(x, \text{tx})$
- 5  $\lambda x. \text{city}(x) \wedge \text{large}(x) \wedge \exists y[\text{state}(y) \wedge \text{borders}(y, \text{tx}) \wedge \text{in}(x, y)]$

Item 5 is the point: a relative clause introduces a fresh variable and an existential. No template could have produced it, but composition does – one word at a time.

## The Pattern in Items 2–4

Every one of those had the same shape:

$$\lambda x. \underbrace{\text{noun}(x)}_{\text{restrictor}} \wedge \underbrace{\text{modifier}(x)}_{\text{filter}}$$

Two one-place predicates, conjoined. But recall the type rule:  $\langle e, t \rangle$  cannot *apply* to  $\langle e, t \rangle$ .

### The second composition rule

When two constituents are both of type  $\langle e, t \rangle$ , combine them by conjunction:  $\lambda x. f(x) \wedge g(x)$ . Formal semantics calls this **predicate modification**; in database terms it is a conjunctive WHERE clause.

So the fragment needs *two* composition rules, not one: application, and conjunction of like-typed predicates. That is enough for a surprisingly large fragment of query language.

## Why Bother With Types?

Nothing so far stops us combining things nonsensically:

$\text{tx(ok)} \quad (\lambda x.\text{state}(x))(\lambda y.\text{river}(y))$

A parser that generates candidate derivations will produce garbage like this by the thousand. We need a cheap filter.

### Semantic types

- $e$  – an entity (Texas, Oklahoma, a row in the database)
- $t$  – a truth value (a query that returns yes/no)
- $\langle \sigma, \tau \rangle$  – a function from things of type  $\sigma$  to things of type  $\tau$

**Rule:** you may apply  $\alpha$  to  $\beta$  only if  $\alpha : \langle \sigma, \tau \rangle$  and  $\beta : \sigma$ . Everything else is rejected before it is ever scored.

# The Type Table

| Word class        | Example        | Type                                                                            | Lambda term                                               |
|-------------------|----------------|---------------------------------------------------------------------------------|-----------------------------------------------------------|
| Proper noun       | <i>Texas</i>   | $e$                                                                             | $tx$                                                      |
| Common noun       | <i>state</i>   | $\langle e, t \rangle$                                                          | $\lambda x. \text{state}(x)$                              |
| Intransitive verb | <i>flows</i>   | $\langle e, t \rangle$                                                          | $\lambda x. \text{flows}(x)$                              |
| Adjective         | <i>large</i>   | $\langle e, t \rangle$                                                          | $\lambda x. \text{large}(x)$                              |
| Transitive verb   | <i>borders</i> | $\langle e, \langle e, t \rangle \rangle$                                       | $\lambda y. \lambda x. \text{borders}(x, y)$              |
| Determiner        | <i>every</i>   | $\langle \langle e, t \rangle, \langle \langle e, t \rangle, t \rangle \rangle$ | $\lambda P. \lambda Q. \forall x [P(x) \rightarrow Q(x)]$ |
| Preposition       | <i>in</i>      | $\langle e, \langle e, t \rangle \rangle$                                       | $\lambda y. \lambda x. \text{in}(x, y)$                   |
| Sentence          | —              | $t$                                                                             | $\text{borders}(tx, \text{ok})$                           |

## Two things worth noticing.

- Nouns, intransitive verbs, and adjectives **share a type** – all are one-place predicates, or table filters.
- Determiners take two predicates and return a truth value, which is why quantification is where semantic parsers usually break.

## Types as a Parser Filter

A CKY-style semantic parser builds constituents bottom-up. At each cell it asks: *can these two combine?*

| Left                                      | Right                  | Result                                 |
|-------------------------------------------|------------------------|----------------------------------------|
| $\langle e, t \rangle$                    | $e$                    | $t$ ✓                                  |
| $\langle e, \langle e, t \rangle \rangle$ | $e$                    | $\langle e, t \rangle$ ✓               |
| $e$                                       | $\langle e, t \rangle$ | ✗ – entities are not functions         |
| $\langle e, t \rangle$                    | $\langle e, t \rangle$ | ✗ by application (but see conjunction) |

The type system prunes the search space **before** any statistical model is consulted – for a lexicon with several entries per word, the difference between a tractable chart and an intractable one.

This is also why type-driven parsing survived into the neural era: it is a hard constraint, and hard constraints are cheap.

# Combinatory Categorical Grammar

Types tell us *what* can combine. They do not tell us about **word order**.

$\lambda y.\lambda x.$  borders( $x, y$ ) says the verb takes two entities – but not that the object is on the right and the subject on the left.

## CCG's idea

Put the directionality *in the category*. A category is either atomic ( $S$ ,  $NP$ ,  $N$ ) or a slash category:

- $X/Y$  – “an  $X$ , if you give me a  $Y$  **on the right**”
- $X\backslash Y$  – “an  $X$ , if you give me a  $Y$  **on the left**”

The slash always leans toward the argument. So  $S\backslash NP$  is a verb phrase: give it a noun phrase on its left and you get a sentence.

CCG is the standard syntax-semantics interface in NLP – Steedman's framework, and the basis of most semantic parsing work from Zettlemoyer & Collins (2005) onward.

## Categories and Types, Side by Side

| Word           | CCG category                  | Semantic type                                                                   | Match? |
|----------------|-------------------------------|---------------------------------------------------------------------------------|--------|
| <i>Texas</i>   | $NP$                          | $e$                                                                             | ✓      |
| <i>state</i>   | $N$                           | $\langle e, t \rangle$                                                          | ✓      |
| <i>flows</i>   | $S \backslash NP$             | $\langle e, t \rangle$                                                          | ✓      |
| <i>borders</i> | $(S \backslash NP) / NP$      | $\langle e, \langle e, t \rangle \rangle$                                       | ✓      |
| <i>every</i>   | $(S / (S \backslash NP)) / N$ | $\langle \langle e, t \rangle, \langle \langle e, t \rangle, t \rangle \rangle$ | ✓      |

The category and the type carry the **same arity information**. The category adds *direction*; the type adds *what kind of thing* each argument is. Neither subsumes the other – which is why CCG lexical entries always list both.

Strip the slashes and replace  $NP$  by  $e$ ,  $S$  by  $t$ : you recover the semantic type exactly.

## Two Combinatory Rules

Forward application ( $>$ )

$$X/Y : f \quad Y : a \quad \Longrightarrow \quad X : f(a)$$

Backward application ( $<$ )

$$Y : a \quad X \backslash Y : f \quad \Longrightarrow \quad X : f(a)$$

Two rules. In both cases the semantics is **function application** – the slash direction only says where to look for the argument.

This is the rule-to-rule hypothesis in its cleanest form: syntax and semantics are the same derivation, read two ways.

Full CCG adds composition (**B**), type-raising (**T**), and substitution (**S**). We need only the two application rules today; type-raising reappears when we get to quantifier scope.

## A CCG Derivation

| <i>Texas</i>                                                | <i>borders</i>                               | <i>Oklahoma</i> |
|-------------------------------------------------------------|----------------------------------------------|-----------------|
| $NP$                                                        | $(S \backslash NP) / NP$                     | $NP$            |
| tx                                                          | $\lambda y. \lambda x. \text{borders}(x, y)$ | ok              |
| <hr/>                                                       |                                              |                 |
| $S \backslash NP : \lambda x. \text{borders}(x, \text{ok})$ |                                              | >               |
| <hr/>                                                       |                                              |                 |
| $S : \text{borders}(\text{tx}, \text{ok})$                  |                                              | <               |

- ① **Forward application:**  $(S \backslash NP) / NP$  finds its  $NP$  on the right. Semantically, apply the verb to the object.
- ② **Backward application:**  $S \backslash NP$  finds its  $NP$  on the left. Apply the VP to the subject.

The syntactic derivation and the semantic derivation are the same object. Read the top line for structure, the bottom for meaning.

## Two More Combinators

Application alone cannot derive every constituent we need. CCG adds two more rules.

Forward composition (B)

$$X/Y : f \quad Y/Z : g \quad \Longrightarrow \quad X/Z : \lambda x. f(g(x))$$

Type-raising (T)

$$X : a \quad \Longrightarrow \quad T/(T \setminus X) : \lambda f. f(a)$$

**Type-raising should look familiar.**  $\lambda f. f(a)$  turns an entity into “the set of properties it has” – this is Partee’s LIFT from Session 2, wearing a CCG hat.

Together, B and T let CCG treat non-traditional word sequences as constituents, which is what makes coordination and extraction work.

## Why You Need Them: Coordination

*Texas borders Oklahoma and New Mexico*

Fine – *Oklahoma and New Mexico* is a constituent. But now:

*Texas borders and Louisiana borders Oklahoma*

*Texas borders* is not a constituent in traditional phrase structure. With type-raising and composition, it is:

---

|                      |                                                                             |
|----------------------|-----------------------------------------------------------------------------|
| <i>Texas</i>         | $NP \xrightarrow{\mathbf{T}} S/(S \backslash NP) : \lambda f. f(\text{tx})$ |
| <i>borders</i>       | $(S \backslash NP)/NP$                                                      |
| <i>Texas borders</i> | $S/NP$ by forward composition <b>B</b>                                      |

---

Now it has a category and a meaning, so it can be coordinated with another  $S/NP$ . This flexibility about constituency is CCG's main selling point.

## Type-Raising Gives You Scope

Recall the problem from the quantifier slide: with two quantifiers there are two readings.

*Every state borders a state.*

In CCG, type-raising is *optional* and can apply to either argument. Different choices give different derivations – and different scopes.

| Derivation                          | Reading                       |
|-------------------------------------|-------------------------------|
| subject quantifier applied last     | $\forall > \exists$ (surface) |
| object type-raised and applied last | $\exists > \forall$ (inverse) |

**Scope ambiguity becomes derivational ambiguity.** One string, two well-formed derivations, two logical forms.

Elegant – and a practical headache, since the number of derivations grows fast. Session 5 works the formal-semantics version of the same problem.

# Why CCG Won in NLP

- **Radically lexicalized.** Almost all the grammar lives in the lexicon; there are only a handful of universal combinatory rules. *So learning a grammar reduces to learning a lexicon* – which is a much better-posed statistical problem.
- **Transparent interface.** Every syntactic category determines a semantic type. No separate mapping to maintain.
- **Flexible constituency.** Composition and type-raising let CCG derive non-standard constituents, which handles coordination and extraction gracefully – *states that border Texas and flow into the Gulf*.
- **Parseable.** CCGbank exists; the parsers are fast and accurate.

Lineage worth knowing: Zettlemoyer & Collins (2005) learned CCG lexicons for GeoQuery; Kwiatkowski et al. generalized the lexical induction; Artzi & Zettlemoyer applied it to instructions and dialogue.

## Quantifiers, Briefly

*Every state borders a state.*

every has category  $(S/(S \setminus NP))/N$  and denotation

$$\lambda P. \lambda Q. \forall x [P(x) \rightarrow Q(x)]$$

- 1 Combine with the noun:  $\lambda Q. \forall x [\text{state}(x) \rightarrow Q(x)]$  – category  $S/(S \setminus NP)$
- 2 That is a *function looking for a VP on its right*. Apply it to the VP.

Notice what happened: the subject is now the function and the verb phrase is the argument – the reverse of the proper-name case. The category  $S/(S \setminus NP)$  says so explicitly.

And with two quantifiers there are two derivations, giving two scope readings. That is Session 5, and in CCG terms it is handled with type-raising and composition.

# Ambiguity Is Not a Bug Report

A semantic parser must produce *several* logical forms, not one, because sentences genuinely have several readings.

| Type         | Example                             | Source                     |
|--------------|-------------------------------------|----------------------------|
| Lexical      | <i>bank</i>                         | one word, several entries  |
| Attachment   | <i>cities in states with rivers</i> | where the PP attaches      |
| Scope        | <i>every state borders a state</i>  | quantifier order           |
| Coordination | <i>large cities and states</i>      | what <i>large</i> modifies |

**Design consequence.** The grammar's job is to enumerate the *well-formed* readings; a statistical model's job is to rank them; and the execution engine can sometimes break the tie – if only one reading returns rows, that is a strong signal.

Note that this division of labour requires the grammar to be *complete* about what is possible. That is again the fragment discipline: be exact about coverage.

## Negation, and Where It Bites

---

|                                          |                                                                        |
|------------------------------------------|------------------------------------------------------------------------|
| <i>Texas does not border Maine</i>       | $\neg \text{borders}(\text{tx}, \text{me})$                            |
| <i>No state borders Hawaii</i>           | $\neg \exists x [\text{state}(x) \wedge \text{borders}(x, \text{hi})]$ |
| <i>Every state does not border Texas</i> | two readings                                                           |

---

The third is ambiguous between  $\forall x [\dots \rightarrow \neg \dots]$  and  $\neg \forall x [\dots \rightarrow \dots]$  – negation and quantifiers scope over each other just as two quantifiers do.

**Practical warning.** Negation is where executable semantics gets expensive:  $\neg P(x)$  over an open domain means *everything in the database that is not  $P$* . Most systems use a closed-world assumption and hope. It is worth knowing that this is an assumption rather than a theorem.

## A Problem: Adverbs and Adjuncts

*Alice read the book in the library on Tuesday.*

With a fixed-arity predicate we are stuck:

read(alice, book)      read(alice, book, library)      read(alice, book, library, tuesday)

### Three problems.

- We need a different predicate for every number of modifiers – unboundedly many.
- The entailment *Alice read the book in the library*  $\Rightarrow$  *Alice read the book* does not fall out; we would have to stipulate it.
- Modifiers can be added freely and in any order, which arity-fixed predicates cannot model.

This is the *adverbial modification* problem, and it has a classic solution.

# Davidson's Move: Quantify Over Events

Davidson (1967)

Verbs introduce an extra argument – an **event variable**. Modifiers are predicates of that event.

$$\exists e [ \text{read}(e, \text{alice}, \text{book}) \wedge \text{in}(e, \text{library}) \wedge \text{on}(e, \text{tuesday}) ]$$

- Modifiers are now **conjuncts**, so they can be added freely and in any order.
- Dropping a conjunct is valid inference, so the entailment is automatic. ✓
- The verb keeps a fixed arity.

Conjunction is commutative and monotone – exactly the properties adjunct modification has. The representation now matches the phenomenon.

## Neo-Davidsonian: Roles as Separate Predicates

Go one step further and pull the arguments out too:

$$\exists e [ \text{read}(e) \wedge \text{agent}(e, \text{alice}) \wedge \text{theme}(e, \text{book}) \wedge \text{in}(e, \text{library}) ]$$

|                    | Davidsonian  | Neo-Davidsonian |
|--------------------|--------------|-----------------|
| Verb arity         | fixed, $n+1$ | always 1        |
| Arguments          | positional   | named roles     |
| Optional arguments | awkward      | trivial         |

**This is the representation NLP actually uses.** PropBank, FrameNet, and AMR are all essentially neo-Davidsonian: an event node, plus labelled edges to its participants.

And semantic role labelling – one of the standard NLP tasks – is exactly the job of recovering those agent, theme, location predicates from text.

## Why This Matters for the Rest of the Course

Look at the shape of the neo-Davidsonian representation:

$$\exists e [ \text{read}(e) \wedge \text{agent}(e, \text{alice}) \wedge \text{theme}(e, \text{book}) ]$$

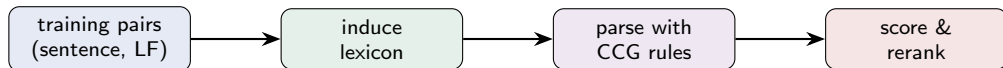
It is a set of **(role, filler) pairs** attached to an event node.

| Here                 | Later in the course             |
|----------------------|---------------------------------|
| role–filler binding  | the binding problem of Part III |
| event node           | the object roles attach to      |
| conjunction of roles | superposition of bindings       |

**Hold onto this.** When Session 11 introduces tensor product and holographic representations, the structure they try to encode in a vector is precisely this – roles bound to fillers, superposed. Neo-Davidsonian semantics is the target; the question is whether a vector space can hold it.

# Where the Lexicon Comes From

We have been writing lexical entries by hand. In practice they are **learned**.



- **Supervision.** Full logical forms (expensive), or just answers – *denotation-only* supervision, where the system searches for any LF that returns the right rows.
- **Ambiguity.** A word gets several candidate entries; a learned model picks among derivations.
- **The types do the pruning**, as we saw – the statistical model only ever sees well-typed candidates.

Zettlemoyer & Collins (2005) is the canonical reference; Liang, Jordan & Klein (2013) is the canonical denotation-only version.

# Evaluating a Semantic Parser

| Metric                     | Compares                     | Note                               |
|----------------------------|------------------------------|------------------------------------|
| Exact match                | LF string vs. gold LF        | brittle; punishes equivalent forms |
| Logical-form accuracy      | LF up to normalization       | better, needs a normalizer         |
| <b>Denotation accuracy</b> | answer set vs. gold answers  | the one usually reported           |
| Execution accuracy         | does the query run and match | standard for text-to-SQL           |

**Why denotation accuracy dominates.** Two logical forms can differ syntactically and be semantically identical:

$$\lambda x. \text{state}(x) \wedge \text{borders}(x, \text{tx}) \quad \equiv \quad \lambda x. \text{borders}(x, \text{tx}) \wedge \text{state}(x)$$

Exact match calls the second wrong; denotation accuracy calls it right.

But denotation accuracy can also be fooled – a wrong logical form that happens to return the right rows on this database scores as correct. This is called *spurious* correctness, and it is a real problem when training with denotation-only supervision.

# Compositional Generalization

The empirical question this whole framework raises: **do learned models compose?**

| Benchmark | Tests                     | Design                                    |
|-----------|---------------------------|-------------------------------------------|
| SCAN      | command → action sequence | train on primitives, test on combinations |
| COGS      | sentence → logical form   | novel structural combinations at test     |
| CFQ       | question → SPARQL         | maximum compound divergence               |

**The construction is always the same.** Train on *jump, walk twice, run left*. Test on *jump twice*. A compositional system generalizes for free; a memorizing system does not.

Note what these benchmarks presuppose: a theory of what the right answer *is* for an unseen combination. That theory is the compositional semantics in this deck.

And the persistent gap on these benchmarks is one motivation for asking, later in this course, whether vector representations can be made to compose properly at all.

# Does Any of This Survive the LLM Era?

An honest question, and worth asking directly. A large model will answer *What states border Texas?* without any of this machinery.

**What has changed.** Hand-built grammars are no longer how you ship a question-answering system.

**What has not.**

- **Executable semantics.** Text-to-SQL, tool calls, and API argument-filling are semantic parsing under different names – and the target is still a structured, executable form.
- **Verifiability.** A logical form can be checked against a database. A generated answer cannot.
- **Compositional generalization.** The benchmarks that test it are built on exactly this theory.
- **The evaluation standard.** To ask whether a model composes, you need a theory of what composing *is*.

That last point is why this material opens the course rather than sitting in an appendix.

# Summary

| Concept                | Key point                                                                 |
|------------------------|---------------------------------------------------------------------------|
| Meaning representation | unambiguous, executable, compositional, inference-supporting              |
| Logical form           | FOL; predicates are tables, constants are keys                            |
| The model              | your database – “true” means “the row is there”                           |
| Why not templates      | language nests without bound                                              |
| Lambda abstraction     | a formula with a labelled gap                                             |
| Partial application    | each step yields a real constituent                                       |
| Semantic types         | $e, t, \langle \sigma, \tau \rangle$ ; a cheap hard filter on derivations |
| CCG category           | arity plus <i>direction</i> ; strips to the semantic type                 |
| Forward/backward app.  | two rules; semantics is application either way                            |
| Lexicalization         | grammar lives in the lexicon, so it can be learned                        |
| Davidsonian events     | modifiers as conjuncts; entailments fall out                              |
| Neo-Davidsonian        | event node plus named roles – PropBank, AMR                               |

# Reading and Connections

## Primary:

- Jurafsky & Martin, *Speech and Language Processing* (3rd ed.), chapters on logical representations of meaning and semantic role labelling
- Steedman, *The Syntactic Process* (2000) – CCG, for the ambitious

## Classic papers, if you want them:

- Zelle & Mooney (1996) – GeoQuery
- Zettlemoyer & Collins (2005) – learning CCG for semantic parsing
- Davidson (1967) – *The Logical Form of Action Sentences*

## How this connects to the main track:

- Session 2 covers the same types and composition, from formal semantics
- Session 3 makes “the database is the model” precise
- Session 5 handles the quantifier scope this deck deferred
- Session 11 asks whether the role–filler structure above fits in a vector

# Questions?

Companion deck — main track resumes with Session 3