

λ -DCS: Dependency-Based Compositional Semantics

The Logical Tradition at Scale — and What It Still Cannot Do

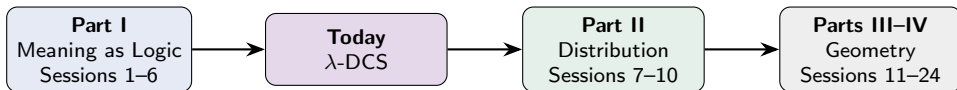
James Pustejovsky

Brandeis University
Department of Computer Science

CS 135: Computational Semantics

Fall 2026 (Session 6A — closing Part I)

Where This Sits



Part I gave us a semantics that is exact, compositional, and truth-conditional. Today we look at what happens when you push that tradition as hard as it will go **computationally** – millions of facts, real questions, learned lexicons.

The point of placing this here: λ -DCS is the *strongest* version of the symbolic approach. When Part II opens with the similarity problem next week, you should not be able to think “a better logical formalism would fix that”. **This is the better logical formalism.**

Today

- ① The problem: logical forms are unwieldy for a parser
- ② λ -DCS: denotations as sets, variables made implicit
- ③ The constructions – join, intersection, aggregation, superlatives
- ④ Exact correspondence with the lambda calculus
- ⑤ Why compactness makes learning possible
- ⑥ **An honest assessment** of what it bridges and what it does not
- ⑦ Join as contraction – a real link forward to Part III

Reference: Liang, Jordan & Klein, *Learning Dependency-Based Compositional Semantics* (2013); Liang, *Lambda Dependency-Based Compositional Semantics* (2013).

A Realistic Query

Which scientists born in Seattle won a Nobel Prize?

In the lambda calculus of Sessions 2–4, over a knowledge base like Freebase:

$$\begin{aligned} \lambda x. \exists p [& \text{Profession}(x, p) \wedge p = \text{Scientist}] \\ & \wedge \exists s [\text{PlaceOfBirth}(x, s) \wedge s = \text{Seattle}] \\ & \wedge \exists a [\text{AwardsWon}(x, a) \wedge \exists t [\text{AwardType}(a, t) \wedge t = \text{NobelPrize}]] \end{aligned}$$

Correct, and entirely standard. It is also **four existentials, four bound variables, and three equalities** – for a question a person would ask in eight words.

The semantics is fine. The problem is what this costs a *system* that has to construct it.

Three Costs

- 1 **Verbosity.** The logical form is much longer than the sentence. If a parser generates it token by token, most of what it emits is bookkeeping – $\exists$, variable names, conjunctions – rather than content.

- 2 **Spurious ambiguity.** These are the same formula:

$$\exists y[R(x, y) \wedge S(y)] \quad \exists z[R(x, z) \wedge S(z)] \quad \exists y[S(y) \wedge R(x, y)]$$

A search procedure does not know that. It explores all of them, and a learned model must be taught they are equivalent.

- 3 **Search space.** Semantic parsing means *searching* over derivations. Every extra degree of freedom – variable naming, conjunct ordering, quantifier placement – multiplies the space without adding meaning.

None of this is a problem for a semanticist working by hand. All of it is a problem for a parser that must consider ten thousand candidates per sentence.

What We Would Want

A representation language that:

- is **compact** – roughly the size of the sentence, not five times it;
- has **no free variable naming**, so equivalent meanings have one form;
- is **compositional**, so it can still be built piece by piece;
- is **directly executable** against a knowledge base;
- and whose **tree structure resembles the sentence's**, so the parser's job is closer to a syntactic one.

λ -DCS is a language designed against exactly this list.

Its ancestry: description logics (which have implicit variables and are built for querying), and dependency syntax (whose trees are flatter and closer to word order than constituency trees). The name records both debts.

The Central Move: Denotations Are Sets

In the lambda calculus, a one-place predicate denotes a function $\mathcal{D} \rightarrow \{0, 1\}$, and a sentence denotes a truth value.

In λ -DCS, a **unary** denotes a *set of entities* directly.

λ -DCS	Denotes	Lambda calculus
Seattle	$\{\text{Seattle}\}$	$\lambda x. x = \text{Seattle}$
Scientist	all scientists	$\lambda x. \text{Scientist}(x)$
PlaceOfBirth	set of <i>pairs</i>	$\lambda x. \lambda y. \text{PlaceOfBirth}(x, y)$

- A **unary** is a set of entities – type $\langle e, t \rangle$ in disguise.
- A **binary** is a set of pairs – type $\langle e, \langle e, t \rangle \rangle$ in disguise.

Nothing has changed semantically. What changes is the *notation* – and notation is exactly what was costing us.

Join: The Core Operation

Join

For a binary B and a unary U :

$$B.U = \{ x : \exists y (x, y) \in B \wedge y \in U \}$$

Example.

$$\text{PlaceOfBirth.Seattle} = \{ x : x \text{ was born in Seattle} \}$$

In the lambda calculus this is

$$\lambda x. \exists y [\text{PlaceOfBirth}(x, y) \wedge y = \text{Seattle}]$$

The existential quantifier has not disappeared – it has become implicit in the operation. Every join carries one $\exists$, and since joins are the only way to traverse a relation, we never write $\exists$ again.

Chained Joins

Joins compose, and each link adds an implicit existential.

PlacesLived.Location.Seattle

denotes people who have lived in Seattle – traversing from a person, to a “places lived” event node, to its location.

The lambda-calculus equivalent:

$$\lambda x. \exists y [\text{PlacesLived}(x, y) \wedge \exists z [\text{Location}(y, z) \wedge z = \text{Seattle}]]$$

Three symbols, against fourteen.

And the λ -DCS version reads left to right, in the order the relation is traversed – which is close to the order the English says it.

Intersection, Union, Negation

Operation	λ -DCS	Denotes
Intersection	$U_1 \sqcap U_2$	$\llbracket U_1 \rrbracket \cap \llbracket U_2 \rrbracket$
Union	$U_1 \sqcup U_2$	$\llbracket U_1 \rrbracket \cup \llbracket U_2 \rrbracket$
Negation	$\neg U$	$\mathcal{D} \setminus \llbracket U \rrbracket$

Worked example. *Scientists born in Seattle:*

$\text{Profession.Scientist} \sqcap \text{PlaceOfBirth.Seattle}$

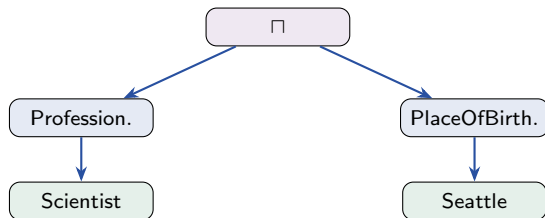
which in the lambda calculus is

$\lambda x. \exists p[\text{Profession}(x, p) \wedge p = \text{Scientist}] \wedge \exists s[\text{PlaceOfBirth}(x, s) \wedge s = \text{Seattle}]$

Intersection is predicate modification from Session 2, written as an operator on sets rather than a rule on functions.

The Tree Structure

λ -DCS formulas are trees, and the tree looks like a dependency parse.



- **Leaves** are entities and types – the content words.
- **Edges** are relations – the joins.
- **Internal nodes** are set operations.

Because every node denotes a *set*, every subtree is independently executable. A parser can evaluate partial derivations and prune those that return nothing – which is a large practical win.

Aggregation

Real questions count, sum, and average.

Question	λ -DCS	Denotes
<i>How many cities are in California?</i>	$\text{count}(\text{Type.City} \sqcap \text{In.CA})$	a number
<i>Total population of Texas cities</i>	$\text{sum}(\text{Population}(\dots))$	a number

A type wrinkle. $\text{count}(U)$ denotes a *number*, not a set of entities – so strictly it denotes the singleton set containing that number.

This keeps the language uniform: everything denotes a set, so everything composes with everything else by the same rules. It is an engineering decision, and a good one – uniformity is what keeps the parser simple.

Note that aggregation is genuinely beyond first-order logic. λ -DCS is not a notational variant of FOL; it is FOL plus counting.

Superlatives

What is the largest city in California?

Superlative

$$\operatorname{argmax}(U, B) = \{ x \in \llbracket U \rrbracket : \llbracket B \rrbracket(x) \geq \llbracket B \rrbracket(y) \text{ for all } y \in \llbracket U \rrbracket \}$$

$$\operatorname{argmax}(\text{Type.City} \sqcap \text{In.CA}, \text{Population})$$

- First argument: the set to search over.
- Second argument: the binary that scores each element.

Superlatives are a genuine test case. In the lambda calculus they require nested quantification with a comparison ($\forall y [\dots \rightarrow \text{pop}(x) \geq \text{pop}(y)]$), and they are a classic source of parser errors. Here they are one operator.

When You Do Need Variables

Implicit variables cover most cases. Some structures still need an explicit one.

Lambda abstraction builds a binary out of a unary with a hole:

$$\lambda x. (\text{Type.Person} \sqcap \text{PlaceOfBirth}.x)$$

– the relation *person born in* ____.

Mu abstraction ($\mu x. \dots$) binds a variable that is reused in several places – needed when the same entity fills two roles:

Who directed a film they also starred in?

This is the same phenomenon we met in Session 3: genuine variable *reuse* – one entity in two argument positions – cannot be made implicit. Reflexives, and structures where a filler is shared, force the variable back into the open.

Worth filing away: this is the copying problem, and it will come back in Part IV.

Coverage Summary

Construct	Notation	Handles
Unary	Scientist	entity sets, types
Binary	PlaceOfBirth	relations
Join	$B.U$	relation traversal, implicit $\exists$
Intersection	$U_1 \sqcap U_2$	modification, restrictive relatives
Union	$U_1 \sqcup U_2$	disjunction
Negation	$\neg U$	complement (closed-world)
Aggregation	count, sum	counting questions
Superlative	argmax, argmin	<i>largest, most, first</i>
Lambda	$\lambda x. \dots$	building binaries
Mu	$\mu x. \dots$	variable reuse

Ten constructs. This covers the overwhelming majority of factoid questions people actually ask of a knowledge base – which is the design target, and a much narrower target than “natural language”.

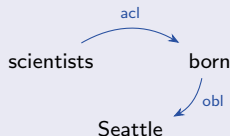
Two Ancestries

Description logic

$\text{Scientist} \sqcap \exists \text{bornIn}.\{\text{Seattle}\}$

- Concepts denote *sets*
- Roles denote *relations*
- Existential restriction = join
- Built for **querying**, and decidable

Dependency syntax



- Flat, head-and-dependent
- Edges are *relations*
- Closer to word order than constituency

λ -DCS takes the *semantics* from the left and the *tree shape* from the right – the formula has almost the shape of the dependency parse, which is what makes parsing easier.

Hence “dependency-based” in the name. The λ is for the abstraction operator, added to the earlier variable-free DCS.

A Longer Worked Example

What is the largest city in a state bordering Texas?

Build it up.

states bordering Texas : $\text{Type.State} \sqcap \text{Borders.Texas}$

cities in those states : $\text{Type.City} \sqcap \text{In.}(\dots)$

the largest : $\text{argmax}(\dots, \text{Population})$

Assembled:

$\text{argmax}\left(\text{Type.City} \sqcap \text{In.}(\text{Type.State} \sqcap \text{Borders.Texas}), \text{Population}\right)$

Nine symbols for a ten-word question, with one level of nesting per level of linguistic embedding.

Write the lambda-calculus equivalent as an exercise. It runs to three lines and five bound variables.

Practice: Translate Into λ -DCS

Predicates available: Type, In, Borders, Population, Capital, Profession, PlaceOfBirth.

- 1 cities in Texas
- 2 the capital of Texas
- 3 states bordering Texas or Oklahoma
- 4 how many states border Texas
- 5 scientists not born in Seattle

Practice: Translate Into λ -DCS

Predicates available: Type, In, Borders, Population, Capital, Profession, PlaceOfBirth.

- 1 cities in Texas
- 2 the capital of Texas
- 3 states bordering Texas or Oklahoma
- 4 how many states border Texas
- 5 scientists not born in Seattle

Answers.

- 1 $\text{Type.City} \sqcap \text{In.Texas}$
- 2 Capital.Texas
- 3 $\text{Type.State} \sqcap (\text{Borders.Texas} \sqcup \text{Borders.Oklahoma})$
- 4 $\text{count}(\text{Type.State} \sqcap \text{Borders.Texas})$
- 5 $\text{Profession.Scientist} \sqcap \neg(\text{PlaceOfBirth.Seattle})$

Item 5 is the one to think about: $\neg$ here means *not in the set of Seattle-born people*, under a closed-world assumption. Under an open-world reading it would also include people whose birthplace is simply unrecorded

The Translation, Systematically

λ -DCS	Lambda calculus
U (unary)	$\lambda x. \varphi(x)$
B (binary)	$\lambda x. \lambda y. \psi(x, y)$
$B.U$	$\lambda x. \exists y [\psi(x, y) \wedge \varphi(y)]$
$U_1 \sqcap U_2$	$\lambda x. \varphi_1(x) \wedge \varphi_2(x)$
$U_1 \sqcup U_2$	$\lambda x. \varphi_1(x) \vee \varphi_2(x)$
$\neg U$	$\lambda x. \neg \varphi(x)$
$\lambda x. U$	$\lambda y. \lambda x. \varphi(x, y)$

A complete, mechanical translation. λ -DCS is not a different semantics – it is the same semantics, in a notation where the common patterns are short and the variable bookkeeping disappears.

Modulo aggregation and superlatives, which take you beyond first-order logic proper.

What Is Gained

	Lambda calculus	λ -DCS
Length vs. sentence	3–5×	$\approx 1\times$
Variable naming	explicit	implicit
Spurious ambiguity	high	low
Partial forms executable	no	yes
Tree shape	constituency-like	dependency-like

The fourth row deserves emphasis. In λ -DCS **every subtree denotes a set**, so a parser can execute a partial derivation and check whether it returns anything.

A candidate that already denotes the empty set can be pruned before it is ever completed. In the lambda calculus, a partial derivation is a function with a free variable – there is nothing to run.

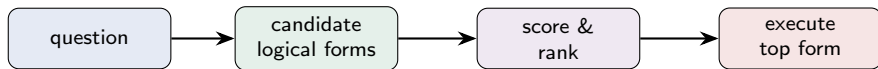
What Is Not Gained

Be clear about this: λ -DCS is no more expressive than what it translates into.

- It does not handle any phenomenon the lambda calculus cannot.
- It does not resolve scope ambiguity – it inherits it. In practice, systems built on λ -DCS often simply do not generate the inverse-scope reading.
- It does not solve intensionality, tense, modality, or plurals; these are outside the target domain.
- Its negation is closed-world, which is a database convention, not a semantics.

The contribution is **engineering**, in the honorable sense: same expressive power, radically better constants. That is exactly what you want when the bottleneck is search rather than expressiveness.

Semantic Parsing as Search



The second box is the expensive one. Its size is governed by:

- how many lexical entries each word has;
- how many ways constituents can combine;
- and **how many notational variants of the same meaning exist.**

λ -DCS attacks the third directly, and helps with the second by making partial forms executable.

This is why the formalism was invented by people building systems rather than by people doing semantics. The motivation is entirely computational.

Weak Supervision: Learning From Answers

The bigger consequence. Annotating logical forms is expensive and requires experts.

Supervision	Training data	Cost
Full	(question, logical form)	expert annotation
Denotation-only	(question, <i>answer</i>)	crowdsourceable

With denotation-only supervision, the system **searches** for any logical form that returns the correct answer, and treats that as a latent variable.

This is only feasible if execution is cheap and partial forms can be pruned – both of which λ -DCS provides.

And this is the real bridge. Not to distributional semantics, but from formal semantics to **statistical learning** – because it made the logical form a *latent* object that could be learned from cheap supervision.

The Lineage

Work	Dataset	Contribution
Zelle & Mooney 1996	GeoQuery	the task
Zettlemoyer & Collins 2005	GeoQuery	learned CCG lexicons
Liang, Jordan & Klein 2011	GeoQuery	DCS; latent logical forms
Liang 2013	—	λ -DCS
Berant et al. 2013	WebQuestions	Freebase scale; SEMPRES
Berant & Liang 2014	WebQuestions	paraphrasing as a bridge

The scaling story. GeoQuery has ~ 800 facts; Freebase has $\sim 10^9$. Getting from one to the other required the lexicon to be learned, the search to be pruned aggressively, and the logical form to be small.

The unglamorous but decisive problem at Freebase scale was the *lexicon*: mapping “*where was X born*” onto the right one of thousands of relations. Alignment from a text corpus, and paraphrase models, were the answers – statistics doing the work formal semantics could not.

The Bridging Problem

The hard part at Freebase scale is not composition. It is the **lexicon**.

Where was Obama born?

Which of Freebase's thousands of relations does “*born*” map to?

people.person.place_of_birth	probably this
people.person.date_of_birth	wrong type
people.place_lived.location	related but wrong
people.person.nationality	plausible, wrong

Two standard answers:

- **Alignment.** Mine a corpus for phrases co-occurring with known entity pairs, and align them to relations.
- **Paraphrasing.** Generate canonical utterances for candidate forms; score how well each paraphrases the question (Berant & Liang 2014).

Both are *statistical*, and both operate on text, outside the semantic representation. The formalism cannot help here – and that is the shape of the limitation this whole lecture is about.

What About Scope?

λ -DCS inherits the scope problem rather than solving it. So what do real systems do?

Strategy	Consequence
Generate only surface scope	misses inverse readings; usually fine for questions
Generate both, let the ranker choose	larger search space
Use execution to disambiguate	works when only one reading returns rows

Why the first option is defensible here. Factoid questions to a knowledge base are overwhelmingly single-quantifier. *Which cities are in Texas?* has one reading.

This is the fragment discipline again, applied honestly. The system does not claim to cover scope ambiguity; it claims to cover a distribution of questions in which scope ambiguity is rare. Whether that is a limitation or a sensible engineering choice depends entirely on what you are building.

Compared With Query Languages

λ -DCS is not the only compact executable formalism. Why not just use SQL or SPARQL?

	λ -DCS	SPARQL	SQL
Compositional	✓	partial	—
Subtrees executable	✓	—	—
Compact	✓	moderate	verbose
Tracks linguistic structure	✓	—	—
Directly runnable	via compilation	✓	✓

The decisive rows are the middle two. SQL is a fine target and a poor *intermediate* representation: its syntax bears no relation to the sentence's, and you cannot execute half a SELECT.

So: parse to λ -DCS, compile to SPARQL or SQL. The compositional formalism does the linguistic work; the query language does the database work.

λ -DCS and Neural Decoders

Compactness turned out to matter again, for a reason nobody designed for.

When semantic parsing moved to encoder–decoder models, the logical form became a **sequence to generate**. And then:

- shorter sequences are easier to generate correctly;
- fewer notational variants means less arbitrary variation for the model to memorize;
- executable prefixes allow constrained decoding – rejecting continuations that cannot lead to a valid form.

The properties that made λ -DCS good for chart-based search made it good for neural decoding too. That is a sign the design was tracking something real about the task rather than about the algorithm.

The same reasoning explains the current preference for compact, canonical intermediate forms in text-to-SQL and tool-calling systems.

Practice: Read Back Into Logic

Translate each λ -DCS formula into the lambda calculus, making every existential explicit.

- 1 Capital.Texas
- 2 Type.City $\sqcap$ In.Texas
- 3 In.In.USA

Practice: Read Back Into Logic

Translate each λ -DCS formula into the lambda calculus, making every existential explicit.

- 1 Capital.Texas
- 2 Type.City $\sqcap$ In.Texas
- 3 In.In.USA

Answers.

1. $\lambda x. \exists y [\text{Capital}(x, y) \wedge y = \text{Texas}]$
2. $\lambda x. \exists t [\text{Type}(x, t) \wedge t = \text{City}] \wedge \exists s [\text{In}(x, s) \wedge s = \text{Texas}]$
3. $\lambda x. \exists y [\text{In}(x, y) \wedge \exists z [\text{In}(y, z) \wedge z = \text{USA}]]$

Item 3 is the one worth dwelling on: two joins, two existentials, and a genuine two-step traversal – *things inside something inside the USA*. Three symbols became fourteen.

Counting the Cost, Concretely

Query	λ -DCS	Lambda calculus
capital of Texas	3	9
cities in Texas	6	18
things inside things in the USA	5	14
largest city in a bordering state	11	38

Symbol counts, roughly; the exact numbers depend on conventions.

Consistently a factor of three or so. That does not sound dramatic until you remember what it is a factor of three *in*:

- the length of a sequence a decoder must emit without error;
- the depth of a search tree;
- the number of positions at which two equivalent forms can differ.

Search spaces are exponential in depth. A constant factor in the exponent is not a constant factor in the work

The Lasting Lesson, Independent of λ -DCS

Whether or not you ever use this formalism, three things generalize.

- 1 **Notation is not neutral.** Two representations can be semantically identical and computationally worlds apart. Which one you pick determines what is learnable.
- 2 **Make partial results meaningful.** λ -DCS's best property is that every subtree denotes something you can execute. Any representation with that property supports pruning, and most do not have it by accident.
- 3 **Push the implicit into the operation.** The existential quantifier did not vanish – it moved into the definition of join. That move, of absorbing bookkeeping into an algebraic operation, is exactly what we will see again when contraction absorbs argument-passing in Part IV.

The third point is why this lecture sits in this course and not only in an NLP course.

So What Does It Bridge?

Claim	Verdict
Bridges formal semantics and <i>machine learning</i>	Yes
Bridges hand-built and <i>learned</i> lexicons	Yes
Bridges symbolic forms and <i>neural generation</i>	Yes
Bridges logical and <i>distributional</i> meaning	No

The last row is the one to sit with.

In λ -DCS, Scientist and Physicist are two unrelated symbols. Their denotations may overlap in the knowledge base, but **the representation itself carries no notion of similarity** – exactly as in Session 2's lexicon.

Whatever similarity a λ -DCS-based system has comes from *elsewhere* – from embeddings used to align text to relations, sitting outside the semantic representation and feeding it.

The Three-Column Picture

	Lambda calc.	λ -DCS	Distributional
Exact composition	✓	✓	—
Executable	✓	✓	—
Truth conditions	✓	✓	—
Compact	—	✓	✓
Learnable	partial	✓	✓
Similarity	—	—	✓
Graded meaning	—	—	✓

λ -DCS moves two rows – *compact* and *learnable* – from the right column to the left.

It moves neither of the last two. And those are the rows Part II is about. No amount of notational engineering within the symbolic tradition will fill them, because they are not a notation problem.

Why This Is the Right Place to Stop

Part I has now been pushed as far as it goes.

- Sessions 1–4 built the semantics and made it executable.
- Session 5 handled quantification and scope.
- Session 6 surveyed what lies beyond the sentence.
- Today: the same tradition, engineered for scale, learned from data, running against a billion facts.

And *still*, in every one of those systems, the representation of a word is an arbitrary symbol.

The gap is not an implementation detail. It is structural.

Next we work out what each semantic type becomes as a vector, with the semantics still exact — and find that similarity is missing there too, provably. Only then does Part II take the opposite bet: start from similarity, get it for free, and see what happens to composition.

You already know what happens. Session 10 is where we prove it.

One Real Bridge

Join, looked at algebraically.

Join, Rewritten

$$B.U = \{x : \exists y (x, y) \in B \wedge y \in U\}$$

Represent the unary U as a **Boolean vector** $\mathbf{u} \in \{0, 1\}^n$ over the domain, and the binary B as a **Boolean matrix** $M \in \{0, 1\}^{n \times n}$.

Then the join is

$$(B.U)_x = \bigvee_y M_{xy} \wedge u_y$$

That is matrix–vector multiplication, over the Boolean semiring $(\vee, \wedge)$ in place of $(+, \times)$.

The implicit existential quantifier is the *summation index*. $\exists y$ and $\bigvee_y$ are the same operation, written in two traditions.

And Matrix Multiplication Is Contraction

Write it in index notation:

$$r_x = M_{xy} u^y$$

The index y appears twice and is summed over. It is **contracted**.

Tradition	Operation	The shared index
Logic	$\exists y [\psi(x, y) \wedge \varphi(y)]$	bound variable
λ -DCS	$B.U$	implicit in the join
Linear algebra	$M\mathbf{u}$	summation index
Tensor calculus	$M_{xy}u^y$	contracted index
Geometric algebra	$A \lrcorner B$	grade lowered by the contraction

Five names for one operation: combine two structures along a shared position and eliminate that position.

Change the Semiring

Here is the observation to carry into Part III.

λ -DCS join is matrix multiplication over $\{0, 1\}$ with $(\vee, \wedge)$.

What if we use $\mathbb{R}$ with $(+, \times)$ instead?

	Boolean semiring	Real semiring
A unary is	a set	a vector
A binary is	a relation	a matrix
Join is	relation traversal	matrix multiplication
Membership is	yes / no	a degree

The right-hand column has similarity, gradedness, and differentiability – the three things the symbolic tradition cannot give us. And the composition operation is *structurally the same*.

This is the shape of the bet the rest of the course makes. DisCoCat takes it with tensors; Functional Geometric Algebra takes it with graded multivectors and left contraction

The Honest Version of the Bridge

λ-DCS bridges	formal semantics $\rightarrow$ statistical learning
It does not bridge	symbolic $\rightarrow$ distributional <i>meaning</i>
But it exposes	the composition operation shared by both

The third row is why it is worth a lecture in this course.

λ -DCS strips semantic composition down to **join** – one operation, applied over and over, that combines two structures along a shared position. Once you see that this operation is contraction, the question for the rest of the course becomes sharp:

Can we keep the operation and change what it operates on?

Session 11 shows several attempts. Sessions 12–15 build the algebra in which contraction is a first-class citizen. Sessions 17–24 ask whether a semantics can be built on it.

Summary

Concept	Key point
Motivation	lambda LFs are verbose; search, not expressiveness, is the bottleneck
Denotations	unaries are sets, binaries are sets of pairs
Join $B.U$	relation traversal; carries an implicit $\exists$
Intersection $\sqcap$	predicate modification, as a set operation
Aggregation	count, sum – beyond first-order
Superlatives	argmax as one operator, not nested quantification
Mu abstraction	needed exactly when a variable is <i>reused</i>
Translation	complete and mechanical into the lambda calculus
Gain	compactness, low spurious ambiguity, executable subtrees
Real payoff	denotation-only supervision becomes feasible
Limit	no similarity, no gradedness – symbols to the end
Forward	join = Boolean matrix multiplication = contraction

Reading

Primary:

- Liang (2013). *Lambda Dependency-Based Compositional Semantics*. arXiv:1309.4408. Short and readable – start here.
- Liang, Jordan & Klein (2013). Learning dependency-based compositional semantics. *Computational Linguistics* 39(2).

Context:

- Berant, Chou, Frostig & Liang (2013). Semantic parsing on Freebase from question–answer pairs. *EMNLP*.
- Berant & Liang (2014). Semantic parsing via paraphrasing. *ACL*.

If you want the description-logic ancestry: any introduction to $\mathcal{ALC}$ will show you where implicit variables and the join/restriction operators come from.

Possible final project. Take a small knowledge base, implement λ -DCS evaluation over both the Boolean and the real semiring, and measure what changes. That is the last section of this deck, made concrete.

Questions?

Next: Session 6B — Semantic Types and Their Vector Counterparts