

Session 4: The Fragment of English

Lexicon, Composition, and Model Checking

James Pustejovsky

Brandeis University
Department of Computer Science

CS 135: Computational Semantics

September 8, 2026

Where We Were

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

Session 2. Types license composition and determine its direction.

Session 3. Formulas are evaluated against a model by a recursive satisfaction relation – and over a finite domain, that relation is a program.

Today we assemble all three into a working system.

By the end of this session you will have seen a complete, executable fragment of English: a syntax, a lexicon, composition rules, and a model – with every sentence's truth value computable.

This is also the session that teaches the Haskell you need to *write* Problem Set 1, not merely read it.

Today

- ① What a fragment is, and why the discipline matters
- ② The syntax: categories and rules
- ③ The lexicon: form, category, type, denotation
- ④ Composition rules, and the rule-to-rule hypothesis
- ⑤ **Haskell, properly** – signatures, currying, higher-order functions, and how to read a type error
- ⑥ Building the fragment, entry by entry
- ⑦ Scope in the fragment: two readings, in code
- ⑧ What the fragment cannot do

Technical session. Bring the scaffold up on a laptop if you have one.

What Is a Fragment?

Definition

A **fragment** of a natural language is an explicitly specified subset of it, given by four components:

- ① a **syntax** – categories and phrase structure rules;
- ② a **lexicon** – for each word, its category, semantic type, and denotation;
- ③ **composition rules** – one semantic operation for each syntactic rule;
- ④ a **model** – domain and interpretation.

The methodology is Montague's, from *The Proper Treatment of Quantification in Ordinary English* (1973).

The essential move: **do not attempt all of English**. Take a small piece, specify it *completely*, and be exact about what is in and what is out.

Why Fragments?

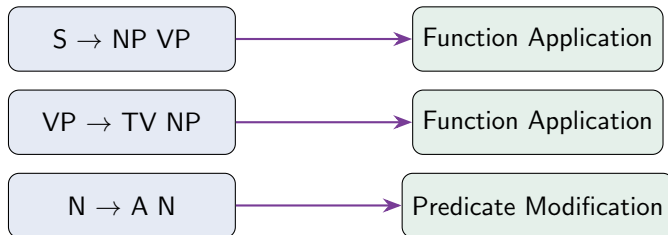
- **Testability.** A fragment makes predictions. Feed it a sentence and a model, and it returns a truth value – which can be right or wrong.
- **Honesty about coverage.** “Our theory handles quantification” is unfalsifiable. “Our fragment derives both readings of *Every student read a book* and neither reading of *Every student read every book they liked*” is a claim.
- **Compositional discipline.** Every rule must pair with a semantic operation. You cannot quietly patch one sentence.
- **Implementability.** A fragment is a specification. If it is precise enough, it compiles.

This is also the standard by which we will judge the geometric framework in Part IV. **Problem Set 3** asks you to rebuild *this fragment* on a geometric substrate – so the comparison is between two implementations of the same specification.

The Rule-to-Rule Hypothesis

Bach (1976)

For every syntactic rule combining constituents, there is exactly one semantic rule that combines their meanings.



Syntax and semantics run in parallel. The semantics is a homomorphic image of the syntax – which is compositionality, stated as an architectural constraint rather than a slogan.

Categories

Our fragment uses eight categories:

Cat.	Name	Type	Examples
S	sentence	t	<i>Alice sleeps</i>
NP	noun phrase	e or $\langle\langle e, t \rangle, t\rangle$	<i>Alice, every student</i>
VP	verb phrase	$\langle e, t \rangle$	<i>sleeps, likes Bob</i>
N	common noun	$\langle e, t \rangle$	<i>student, dog</i>
IV	intransitive verb	$\langle e, t \rangle$	<i>sleeps</i>
TV	transitive verb	$\langle e, \langle e, t \rangle \rangle$	<i>likes, read</i>
DET	determiner	$\langle\langle e, t \rangle, \langle\langle e, t \rangle, t \rangle\rangle$	<i>every, a, no</i>
A	adjective	$\langle e, t \rangle$	<i>tall, sleeping</i>

Note the NP row. Proper names and quantified NPs occupy the same syntactic category but *different* semantic types. We return to this – it is the fragment's one real design tension.

Phrase Structure Rules

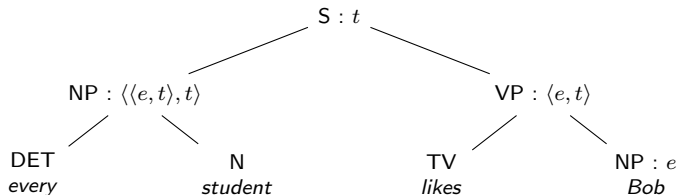
	Rule	Semantic operation	Example
(1)	$S \rightarrow NP VP$	FA	<i>Alice sleeps</i>
(2)	$VP \rightarrow IV$	identity	<i>sleeps</i>
(3)	$VP \rightarrow TV NP$	FA	<i>likes Bob</i>
(4)	$NP \rightarrow PN$	identity	<i>Alice</i>
(5)	$NP \rightarrow DET N$	FA	<i>every student</i>
(6)	$N \rightarrow A N$	PM	<i>sleeping student</i>

Six rules. Every one paired with exactly one semantic operation, as the rule-to-rule hypothesis demands.

“Identity” means the semantics passes through unchanged – the rule is syntactically useful but semantically inert. Some presentations omit such rules entirely.

A Derivation Tree

Every student likes Bob.



Reading bottom-up, with the rule used at each step:

- 1 Rule (5), FA: $\llbracket \text{every} \rrbracket(\llbracket \text{student} \rrbracket)$ – gives a GQ
- 2 Rule (3), FA: $\llbracket \text{likes} \rrbracket(\llbracket \text{Bob} \rrbracket)$ – gives a VP of type $\langle e, t \rangle$
- 3 Rule (1), FA: the GQ applies to the VP – gives t

At step 3 the *subject* is the function and the VP is the argument. At step 2 the verb is the function. The types decide, not the tree geometry.

The NP Tension, Stated

Rule (1) is $S \rightarrow NP\ VP$. But NP has two possible types:

NP	Type	Who applies to whom
<i>Alice</i>	e	VP applies to NP
<i>every student</i>	$\langle\langle e, t \rangle, t\rangle$	NP applies to VP

So rule (1) corresponds to *two* composition paths, depending on the subject – violating the rule-to-rule hypothesis in letter, if not in spirit.

Two responses:

- 1 **Keep both paths.** Simple, and what Problem Set 1 does.
- 2 **Lift every NP to $\langle\langle e, t \rangle, t\rangle$,** using Partee's LIFT from Session 2. Then rule (1) is uniformly “NP applies to VP”. This is Montague's choice.

We take (1) today for clarity. Session 5 gives the argument for (2) – and it is scope that makes the argument.

What This Fragment Covers

In:

- Proper names
- Common nouns
- Intransitive and transitive verbs
- Determiners: *every, a/some, no*
- Intersective adjectives
- Simple declarative sentences
- Scope ambiguity (two quantifiers)

Out:

- Tense and aspect
- Plurals
- Pronouns and anaphora
- Relative clauses
- Coordination
- Negation of sentences
- Modality, attitude verbs
- Non-intersective adjectives

The right-hand column is not a list of failures – it is the fragment being **honest about its scope**. Your PS 1 writeup asks you to name one item from it and say what would have to be added.

Anatomy of a Lexical Entry

Each entry specifies four things:

Form	<i>likes</i>
Category	TV
Type	$\langle e, \langle e, t \rangle \rangle$
Denotation	$\lambda x. \lambda y. \text{like}(y, x)$

The type is not an extra annotation – it is **determined by** the denotation, and **constrains** where the word may appear.

In Haskell the type and the denotation are written on consecutive lines, and the compiler checks that they agree. That checking is the whole reason we are using it.

Names and Nouns

Form	Cat.	Type	Denotation
<i>Alice</i>	PN	e	a
<i>Bob</i>	PN	e	b
<i>student</i>	N	$\langle e, t \rangle$	$\lambda x. \text{student}(x)$
<i>dog</i>	N	$\langle e, t \rangle$	$\lambda x. \text{dog}(x)$
<i>sleeps</i>	IV	$\langle e, t \rangle$	$\lambda x. \text{sleep}(x)$

Note again: *student* and *sleeps* have the same semantic type. Semantically, a common noun and an intransitive verb are the same kind of object – a one-place predicate.

The syntax distinguishes them because they occur in different positions. The semantics does not need to.

Transitive Verbs, and Argument Order

$$\llbracket \text{likes} \rrbracket = \lambda x. \lambda y. \text{like}(y, x) : \langle e, \langle e, t \rangle \rangle$$

The first argument is the object; the second is the subject. Three reasons this is right:

- 1 **Constituency.** English builds VP before S. Applying to the object first yields a VP.
- 2 **Type uniformity.** The result, $\lambda y. \text{like}(y, b)$, has type $\langle e, t \rangle$ – indistinguishable from an intransitive verb. Rule (1) then does not care how the VP was built.
- 3 **Rule economy.** One rule for $S \rightarrow \text{NP VP}$ covers both *Alice sleeps* and *Alice likes Bob*.

This convention is a frequent source of bugs. In the scaffold, likes b a means *a likes b* – object first. Write it on a sticky note.

Determiners

A determiner takes a **restrictor** (the noun) and a **scope** (the VP), and returns a truth value.

$$\text{DET} : \langle \langle e, t \rangle, \langle \langle e, t \rangle, t \rangle \rangle$$

Form	Denotation
<i>every</i>	$\lambda Q. \lambda P. \forall x [Q(x) \rightarrow P(x)]$
<i>a, some</i>	$\lambda Q. \lambda P. \exists x [Q(x) \wedge P(x)]$
<i>no</i>	$\lambda Q. \lambda P. \neg \exists x [Q(x) \wedge P(x)]$

Two structural observations.

- The $\lambda Q. \lambda P$ prefix is identical in all three. Only the logical skeleton differs.
- *every* pairs with $\rightarrow$; *some* and *no* pair with $\wedge$. This is the vacuous-satisfaction point from Session 3, and it is not interchangeable.

Applying a Determiner: *every student*

$$\begin{aligned}\llbracket \text{every student} \rrbracket &= \llbracket \text{every} \rrbracket (\llbracket \text{student} \rrbracket) \\ &= (\lambda Q. \lambda P. \forall x [Q(x) \rightarrow P(x)]) (\lambda z. \text{student}(z)) \\ &\rightsquigarrow \lambda P. \forall x [(\lambda z. \text{student}(z))(x) \rightarrow P(x)] \\ &\rightsquigarrow \lambda P. \forall x [\text{student}(x) \rightarrow P(x)] \quad : \langle \langle e, t \rangle, t \rangle\end{aligned}$$

A **generalized quantifier**: it is waiting for a predicate.

Apply it to *sleeps*:

$$(\lambda P. \forall x [\text{student}(x) \rightarrow P(x)]) (\lambda w. \text{sleep}(w)) \rightsquigarrow \forall x [\text{student}(x) \rightarrow \text{sleep}(x)] : t$$

Which is exactly the formula we evaluated by hand in Session 3 – and found false in our model.

Haskell, Properly

Enough to *write* the fragment, not merely read it.

Everything in this section is notation for something you already know.

Type Signatures

Every definition gets a signature on the line above it.

```
sleeps :: Entity -> Bool
sleeps x = x 'elem' [Carol, Fido]
```

Haskell	Semantic type	Read as
Entity	e	an individual
Bool	t	a truth value
Entity -> Bool	$\langle e, t \rangle$	a one-place predicate
Entity -> Entity -> Bool	$\langle e, \langle e, t \rangle \rangle$	a two-place predicate
(Entity -> Bool) -> Bool	$\langle \langle e, t \rangle, t \rangle$	a generalized quantifier

$\rightarrow$ associates to the *right*: $a \rightarrow b \rightarrow c$ means $a \rightarrow (b \rightarrow c)$.

Which is currying, spelled in the type. A two-argument function *is* a one-argument function returning a function.

Two Ways to Write a Function

These are the same definition:

```
-- equation style
sleeps :: Entity -> Bool
sleeps x = x 'elem' [Carol, Fido]

-- lambda style
sleeps :: Entity -> Bool
sleeps = \x -> x 'elem' [Carol, Fido]
```

$\backslash$ is λ . The second form is literally $\lambda x. x \in \{\text{Carol}, \text{Fido}\}$.

Use whichever is clearer. In the scaffold you will often want lambda style, because a denotation is a lambda term and writing it that way keeps the correspondence visible.

Backticks turn a two-argument function into an infix operator: `x 'elem' xs` is `elem x xs`. Purely cosmetic, but it reads better.

Application Is Juxtaposition

No parentheses needed for application. Application associates to the **left**.

```
likes :: Entity -> Entity -> Bool
```

```
likes Bob          -- partial application: type Entity -> Bool
```

```
likes Bob Alice    -- means (likes Bob) Alice : type Bool
```

Expression	Type	Meaning
likes	Entity -> Entity -> Bool	the relation
likes Bob	Entity -> Bool	<i>likes Bob</i> – the VP!
likes Bob Alice	Bool	<i>Alice likes Bob</i>

The middle row is the point: partial application produces the verb phrase, and its type is exactly that of an intransitive verb. Currying and constituency, lined up.

Object first. *likes Bob Alice* is *Alice likes Bob*, not *Bob likes Alice*.

Higher-Order Functions

Functions are ordinary values: they can be arguments and results.

```
-- takes a predicate, returns a Bool: this is a GQ
someDog :: (Entity -> Bool) -> Bool
someDog p = any (\x -> dog x && p x) domain

-- takes two predicates, returns a Bool: this is a determiner
every :: (Entity -> Bool) -> (Entity -> Bool) -> Bool
every q p = all (\x -> not (q x) || p x) domain
```

- `all :: (a -> Bool) -> [a] -> Bool` – true if the predicate holds of every element
- `any :: (a -> Bool) -> [a] -> Bool` – true if it holds of at least one

These are $\forall$ and $\exists$ over the finite domain – the satisfaction clauses of Session 3, transcribed.

And `not (q x) || p x` is material implication, $Q(x) \rightarrow P(x)$.

Type Synonyms Keep It Readable

Writing `(Entity -> Bool) -> (Entity -> Bool) -> Bool` everywhere is unbearable. The scaffold defines synonyms:

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

Now signatures read like the semantics:

```
student :: ET
likes   :: TV
every   :: DET
```

`type` declares a synonym, not a new type – `ET` and `Entity -> Bool` are interchangeable everywhere. The compiler will sometimes report an error using one and sometimes the other.

Reading a Type Error, I

Suppose you write:

```
s :: T
s = alice sleeps
```

intending *Alice sleeps*. GHC replies:

```
Fragment.hs:87:5: error:
    * Couldn't match expected type: (Entity -> Bool) -> t0
        with actual type: Entity
    * The function 'alice' is applied to one value argument,
        but its type 'Entity' has none
    In the expression: alice sleeps
```

Decode it.

- *actual type*: *Entity* – *alice* is an individual, type *e*.
- *applied to one value argument, but its type has none* – individuals are not functions. They cannot be applied to anything.

The fix is semantic, not syntactic: the predicate applies to the individual, so write *sleeps alice*.

Reading a Type Error, II

Now a subtler one. You write:

```
tallStudent :: ET  
tallStudent = tall student
```

```
Fragment.hs:94:20: error:  
  * Couldn't match expected type: Entity  
    with actual type: Entity -> Bool  
  * In the first argument of 'tall', namely 'student'
```

Decode it. `tall` has type `ET`, so it expects an `Entity`. You gave it `student`, which is itself an `ET`.

This is exactly the mismatch from Session 2 – $\langle e, t \rangle$ applied to $\langle e, t \rangle$. The compiler has just rediscovered the argument for predicate modification. The fix is not to force the application through; it is to use the other composition rule: `pm tall student`.

The Errors Types Will *Not* Catch

Type checking is a theory of what can combine. It is not a theory of what is *correct*.

```
-- typechecks perfectly; means the wrong thing
aliceLikesBob :: T
aliceLikesBob = likes Alice Bob    -- WRONG: this is "Bob likes Alice"
```

Both arguments are `Entity`; the compiler is satisfied. But the object-first convention means this says *Bob likes Alice*.

Types catch	Types miss
combining incompatible kinds	swapped arguments of the same kind
wrong arity	wrong logical connective ($\wedge$ for $\rightarrow$)
missing arguments	wrong quantifier

This is why the scaffold ships a test suite. Types give you well-formedness; the model gives you truth.

Debugging Strategy

- 1 **Read the type error top down.** The first error is usually the real one; later errors are often cascade damage.
- 2 **Compare expected against actual.** The gap between them tells you which composition step is wrong.

- 3 **Ask the compiler.** Put a hole where you are stuck:

```
myDenotation = _
```

GHC will tell you the type it expects there.

- 4 **Test in GHCi.** `ghci Fragment.hs`, then evaluate pieces:

```
> :t likes Bob
likes Bob :: Entity -> Bool
> student Alice
True
```

- 5 **Run the suite often.** `runghc Fragment.hs` reports TODO, FAIL, or PASS per sentence; unfilled holes never crash the run.

The Scaffold, Top to Bottom

Part	Contents	Status
A	Domain, semantic types, extensions, inSet/inRel	given
B	Lexicon – 2 worked, 7 holes	yours
C	Composition rules – 4 holes	yours
D	Sentences – 1 worked, 7 holes	yours
E	Scope – 4 holes	yours
F	Test harness	given

Work in order. Part C depends on B; D depends on both; E depends on all three.

Twenty-two holes, and in the reference solution every single one is a single line. If a hole is taking you more than three or four lines, you are probably fighting the language rather than doing the semantics – come to office hours.

Writing a Noun

The scaffold gives you one worked example:

```
student :: ET
student = inSet studentSet
```

So the holes follow immediately:

```
dog :: ET
dog = inSet dogSet

book :: ET
book = inSet bookSet
```

Why this works. `inSet :: [Entity] -> ET` turns a list into a characteristic function. Recall Session 2: a set and a function of type $\langle e, t \rangle$ are the same object, and `inSet` is the isomorphism.

Equivalent, if you prefer to see the lambda: `dog = \x -> x 'elem' dogSet.`

Writing a Transitive Verb

```
read' :: TV  
read' = inRel readPairs
```

where the scaffold defines

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

Trace the argument order carefully.

read' Hamlet Alice	<i>Alice read Hamlet</i>
looks up	(Alice, Hamlet) 'elem' readPairs

The name has a prime because read is already taken by the Prelude. Same for some' and no' – some and no clash with library names.

Writing a Determiner: *every*

The mathematics:

$$\llbracket \text{every} \rrbracket = \lambda Q. \lambda P. \forall x [Q(x) \rightarrow P(x)]$$

The code, given to you as the worked example:

```
every :: DET
every restrictor scope = all (\x -> not (restrictor x) || scope x) domain
```

Logic	Haskell
$\forall x$	<code>all (\x -> ...) domain</code>
$Q(x)$	<code>restrictor x</code>
$\rightarrow$	<code>not (...) ...</code>
$P(x)$	<code>scope x</code>

Line for line. That correspondence is the reason this course uses a functional language.

Writing *some* and *no*

Your turn. The mathematics:

$$\llbracket \text{some} \rrbracket = \lambda Q. \lambda P. \exists x [Q(x) \wedge P(x)]$$

$$\llbracket \text{no} \rrbracket = \lambda Q. \lambda P. \neg \exists x [Q(x) \wedge P(x)]$$

Writing *some* and *no*

Your turn. The mathematics:

$$\llbracket \text{some} \rrbracket = \lambda Q. \lambda P. \exists x [Q(x) \wedge P(x)]$$

$$\llbracket \text{no} \rrbracket = \lambda Q. \lambda P. \neg \exists x [Q(x) \wedge P(x)]$$

```
some' :: DET
some' restrictor scope = any (\x -> restrictor x && scope x) domain

no' :: DET
no' restrictor scope = not (some' restrictor scope)
```

Note that *no'* is defined in terms of *some'*. That is the quantifier duality from Session 3:
 $\neg \exists x \varphi \equiv \forall x \neg \varphi$, put to work.

And note $\wedge$ for *some*, $\rightarrow$ for *no*. Using implication here would make *Some dog sleeps* true whenever anything fails to be a dog.

Composition Rules in Code

```
-- Function application
fa :: (a -> b) -> a -> b
fa f x = f x

-- Predicate modification
pm :: ET -> ET -> ET
pm p q = \x -> p x && q x

-- Subject-predicate, for a proper-name subject
spName :: Entity -> ET -> T
spName x vp = fa vp x

-- Build a VP from a TV and a proper-name object
vpName :: TV -> Entity -> ET
vpName tv obj = fa tv obj
```

`fa` looks trivial – it is just application. That is the point: in a functional language, the fundamental composition rule of semantics is the language's own primitive.

The polymorphic signature $(a \rightarrow b) \rightarrow a \rightarrow b$ makes it a general rule rather than a special case.

Sentences

With the lexicon and rules in place, sentences are one-liners.

```
-- Carol sleeps.                                [worked example]
s1 :: T
s1 = spName carol sleeps

-- Alice likes Bob.
s3 :: T
s3 = spName alice (vpName likes bob)

-- Every student sleeps.
s4 :: T
s4 = every student sleeps

-- Some student likes Fido.
s8 :: T
s8 = some' student (vpName likes Fido)
```

Read `s4` aloud: “every student sleeps”. The code and the English have the same shape, because the determiner is a function of two predicates and English puts them in that order.

Model Checking

The harness evaluates each sentence and compares against the expected value:

```
tests :: [(String, Bool, Bool)]
tests =
  [ ("D3  Every student sleeps      ", False, s4)
  , ("D4  Some dog sleeps           ", True,  s5)
  , ...
  ]
```

Running it:

```
$ runghc Fragment.hs
CS 135 PS1 -- Fragment test suite
-----
PASS  D0  Carol sleeps
FAIL  D3  Every student sleeps      expected False, got True
TODO  D6  Every student read Hamlet
```

TODO = hole unfilled (the harness catches undefined, so it never crashes). FAIL = filled and typechecks, but wrong against the model. PASS = correct.

A FAIL means your denotation is well-formed but says something false about this model – a *semantic* bug, and the interesting kind.

Two Readings, One String

Every student read a book.

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

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

Both use the same lexical items. What differs is **which quantifier is applied first** – equivalently, which one takes the other in its scope.

And here is the useful fact for Problem Set 1: you do not need any new machinery. *every*, *some*, and *read* are enough. You only need to nest them in the two possible orders.

Surface Scope, in Code

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

```
surface :: T
surface = every student (\x -> some' book (\y -> read' y x))
```

Read it outside in.

- every student (...) – for each student x ...
- some' book (...) – ...there is some book y ...
- read' y x – ...such that x read y . (object first!)

The lambda $\backslash x \rightarrow \dots$ is the scope argument of every: a predicate, built on the fly, which is true of an individual just when that individual read some book.

Inverse Scope, in Code

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

```
inverse :: T
inverse = some' book (\y -> every student (\x -> read' y x))
```

The only change: the two quantifiers swapped places. The innermost expression `read' y x` is identical.

<code>every student (\x -> some' book (\y -> ...))</code>	surface
<code>some' book (\y -> every student (\x -> ...))</code>	inverse

Nesting order *is* scope. This is the clearest possible demonstration of why the $\forall\exists$ / $\exists\forall$ asymmetry from Session 3 matters for natural language.

Why They Differ, in Our Model

$\text{read} = \{(\text{Alice}, \text{Hamlet}), (\text{Bob}, \text{Hamlet}), (\text{Carol}, \text{Dune}), (\text{Dave}, \text{Ulysses})\}$

with $\text{student} = \{\text{Alice}, \text{Bob}, \text{Carol}\}$.

Surface – does each student read *some* book?

Alice $\rightarrow$ Hamlet ✓ Bob $\rightarrow$ Hamlet ✓ Carol $\rightarrow$ Dune ✓ **TRUE**

Inverse – is there *one* book every student read?

	Alice	Bob	Carol	
Hamlet	✓	✓	—	FALSE
Dune	—	—	✓	
Ulysses	—	—	—	

No column is complete. The model was built so the two readings come apart – which means you cannot get both right by accident.

The Entailment

One direction only

inverse $\models$ surface

surface $\not\models$ inverse

If one book was read by every student, then certainly every student read some book. The converse fails – and our model is a counter-example to it.

Problem Set 1's writeup asks you about this: does the entailment hold *in general*, or only in this model? Show the reasoning from the satisfaction definition, not from the test output.

Hint: the argument is the same as the $\exists\forall \models \forall\exists$ proof sketch from Session 3, with the restrictor carried along.

What the Fragment Cannot Do

Phenomenon	Example	What is missing
Anaphora	<i>A student came in. He sat.</i>	cross-sentential binding
Relative clauses	<i>the student who slept</i>	abstraction over a gap
Tense	<i>Alice slept</i>	times in the model
Intensionality	<i>Alice believes Bob left</i>	possible worlds
Non-intersective A	<i>former senator</i>	type $\langle\langle e, t \rangle, \langle e, t \rangle\rangle$
Plurals	<i>the students met</i>	plural individuals in $\mathcal{D}$
Coordination	<i>Alice and every student</i>	type-uniform NPs

Each is a known extension with a substantial literature. Sessions 5 and 6 take up the first, fourth, and last of these.

And note the last row: coordination is the argument for lifting all NPs to $\langle\langle e, t \rangle, t\rangle$ – the design choice we deferred earlier. Session 5 makes it.

Summary

Concept	Key point
Fragment	syntax + lexicon + composition rules + model; specified completely
Rule-to-rule	one semantic operation per syntactic rule
Lexical entry	form, category, type, denotation – type determined by denotation
Argument order	TV takes the object first, so partial application yields a VP
Determiners	$\lambda Q.\lambda P$; <i>every</i> with $\rightarrow$, <i>some/no</i> with $\wedge$
NP tension	names are e , quantified NPs are $\langle\langle e, t \rangle, t\rangle$ – two paths for one rule
Haskell types	$\rightarrow$ right-associative; application left-associative
Type errors	<i>expected</i> vs <i>actual</i> names the failing composition step
Type limits	they catch kind errors, not swapped arguments
Scope	nesting order of quantifiers; no new machinery needed
Entailment	inverse $\models$ surface, one direction only

Coming Up

Session 5 (Friday): Quantifiers and scope

- Generalized quantifiers properly: NPs as sets of sets
- Conservativity, monotonicity, and what determiners can mean
- Why all NPs should be lifted to one type
- Where the two scope readings *come from*, given one syntactic structure

Reading: EU 7.1–7.4; the SEP entry on generalized quantifiers is optional but good.

Problem Set 1 is due Friday 25 September. Parts B and C should be comfortable after today; Part E after Session 5.

Lab this week: the scaffold, GHCi, and the holes in Parts B and C. Bring your laptop and your errors.

Questions?

Next: Session 5 — Quantifiers and Scope