6.863J Natural Language Processing
Lecture 16: The meaning of it all, #2

(or #42)

Instructor: Robert C. Berwick
berwick@ai.mit.edu

The Menu Bar

- Administrivia:
  - Lab 4a out April 14

Agenda:

- What does this all mean?
- Frege's principle of compositionality
- Representation and lambda calculus
- Today: beyond simple VPs; properties & database interfaces; adjectives, prepositions, and more…!

Meaning of meaning, redux

- How can we automate process of associating semantic representations w/ expressions of natural language?
- How can we use semantic representations to automate drawing inferences?

Meaning of meaning redux

- Our semantics:
  - Truth conditional – sentence's meaning in the world is what world would have to be like in order for the sentence to be true
  - Compositional – meaning of whole from parts, reflecting way the parts combine
  - Model-theoretic – logical language used distinguished from its interpretation in the domain of 'real world' objects & relations

Barriers to compositionality

- *Ce corps qui s'appelait e qui s'appelle encore le saint empire romain n'etait en aucune maniere ni saint, ni romain, ni empire.*
- This body, which called itself and still calls itself the Holy Roman Empire, was neither Holy, nor Roman, nor an Empire - *Voltaire*

From sentence meanings to phrase meanings – intermediate summary

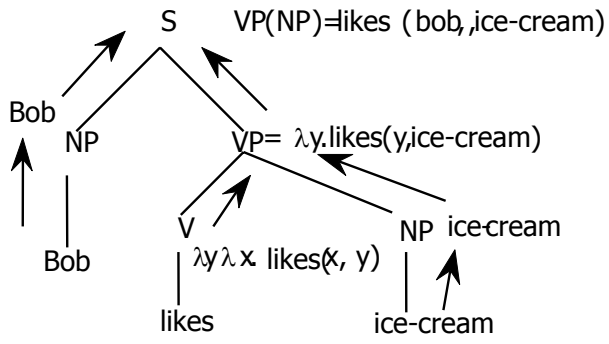
- Sentence meanings are propositions or sets of possible worlds or situations – those situations where the sentence is true
- NP meanings (meanings of proper names) are individuals
- Intransitive verb meanings are functions from individuals to sentence meanings (propositions)
- Transitive verb meanings are functions from individuals to intransitive verb meanings

The story so far

- Simple transitive verb sentence w/ names
- Combine with lambda-calculus to get an output semantic representation, in the same way an intranstive verb combines w/ name:

1. sleep(Bob)
2. love(Bob, ice-cream) or
3. (love :subj bob :obj ice-cream) or – following the textbook:
4. $\exists$e present(e), act(e, loving)$\wedge$lover(e,bob)$\wedge$ lovee(e, ice-cream)

This gives us:



Our logical language, part 1

Three major kinds of objects

1. Booleans
   - Roughly, the semantic values of sentences (T/F)
2. Entities
   - Values of NPs, i.e., objects
   - Maybe also other types of entities, like times
3. Functions of various types
   - A function returning a boolean is called a "predicate" – e.g., frog(x), green(x)
   - Functions might return other functions!
   - Function might take other functions as arguments!

Our secret weapon

- Lambda calculus lets us 'postpone' positions within a FOPC expression and 'fill' them later, by applying the new expression to another
- Fundamental rule for lambda calculus:

Reduction or application (substitute for var):

vp loves ice-cream: $\lambda y\ \lambda x$ love(x,y)bob = λx love (x,bob)

Our logical language, part 2

- Lambda terms:
  - A way of writing "anonymous functions"
    - No function header or function name
    - But defines the key thing: **behavior** of the function
    - Just as we can talk about 3 without naming it "x"
  - Let square = λp p*p
  - Equivalent to int square(p) { return p*p; }
  - But we can talk about λp p*p without naming it
  - Format of a lambda term: λ variable expression

Logic: Lambda Expressions (terms)

- Lambda terms:
  - Let square = λp p*p
  - Then square(3) = (λp p*p)3 = 3*3
  - Note: square(x) isn't a function! It's just the value x*x.
  - But λx square(x) = λx x*x = λp p*p = square
    (proving that these functions are equal – and indeed they are, as they act the same on all arguments: what is (λx square(x))(y)?)
  - Let even = λp (p mod 2 == 0) a predicate: returns true/false
  - even(x) is true if x is even
  - How about even(square(x))?
  - λx even(square(x)) is true of numbers with even squares
    - Just apply rules to get λx (even(x*x)) = λx (x*x mod 2 == 0)
    - This happens to denote the same predicate as even does

Logic: Multiple Arguments

- All lambda terms have one argument
- But we can fake multiple arguments ...
- Claim that times(5)(6) means same as times(5,6)
  - times(5) = (λx λy times(x,y)) (5) = λy times(5,y)
    - If this function weren't anonymous, what would we call it?
  - times(5)(6) = (λy times(5,y))(6) = times(5,6)
- So we can always get away with 1-arg functions ...
  - ... which might return a function to take the next argument. Whoa.
  - We'll still allow times(x,y) as syntactic sugar, though

Our logical language: constants

- ~~Thus, have "constants" that name some of the~~ entities and functions (e.g., times):
  - GeorgeWBush - an entity
  - red – a predicate on entities
    - holds of just the red entities: red(x) is true if x is red!
  - loves – a predicate on 2 entities
    - loves(GeorgeWBush, LauraBush)
    - *Question:* What does loves(LauraBush) denote?
- Constants used to define meanings of words
- Meanings of phrases built from the constants

Our logical language, more formally

- A typed lambda calculus (see notes on web)
- Basic types = {Ind, Bool} (type of individuals, type of propositions)
- The set TYP is the smallest set s.t.
  1. BasTyp $\subseteq$ TYP
  2. If $\sigma, \tau \in$ TYP then $(\sigma \rightarrow \tau) \in$ TYP

From this we define:

VAR_τ = countably infinite set of variables of type τ

CON_τ = set of constants of type τ

Union over these = VAR, CON

Lambda terms: collection of sets, TERM, smallest set built from VAR, CON, and $\alpha(\beta)$, $\lambda x(\alpha)$...

Logical inference

- Once we have the predicate logic forms
- Reason on them directly, using operators $\vee, \wedge, \Rightarrow, \neg$
- Standard approach: relate predicates to one another via implications, e.g.,

  loves(x, y) $\Rightarrow$ likes(x,y)
- These are called Meaning Postulates™

Meaning postulates, II

- Relate verbs & adjectives, e.g.,
  $\forall x, [Open_v (x) \Leftrightarrow Become(Open_A (x))]$

We'll have much more to say about lexical semantics later…

How is all this stuff learned??

Interpretation: from connotation to denotation ("meaning")

- So far, we just have a compositional semantics – Like a programming language though, we must give each statement a model interpretation, ie, a denotation – truth condition in a possible world

Models and meaning

- Model M = {Domain, [[•]] }, a pair of a context frame w/ domains for types (constants, functions…); and an interpretation function [[•]]: CON→ DOM
- This gives us the denotation ("meaning") of semantic expressions (typed lambda forms)

Interpretation: Domains

- Denotation of a constant = an individual
- Denotation of a function = a set
- Denotation of a sentence = a truth value
- Denotation of a variable = an assignment function
- Example model:
  [[bob]] = *bob;* [[ice-cream]] = *ice-cream*
  [[loves]] = {(b,i), (b,b)}

Sentence has truth value of 1 since (b,i) is in [[loves]]
Mapping is not trivial! E.g., [[bob]]=[[ice-cream]]?
Bob = "that person who is referred to as bob"

Interpretation: domains and types

- We use types to make sure the proper arguments go with the proper functions…
- Bob and ice-cream: are IND(ividuals)
- Sleep: is IND→ BOOL (this is a property)
- Love: (do you remember?) love(x,y)
- Love(x,ice-cream) is like 'sleep' – so bob applied to this: IND→ BOOL;
- And 'ice-cream' is IND
- So whole map is: IND→ (IND→ BOOL)

What next? Interpretation = Map to DB world

- Relations (x, y) are 2-way tables
- One table per predicate (often in practice – indirect, e.g., red predicate could come from table w/ objects & colors…)
- Relation love has 2 fields, indicated by arg position, or colon keyword, or by other predicate names
- Names are IDs in the SQL DB: *b* and *i*

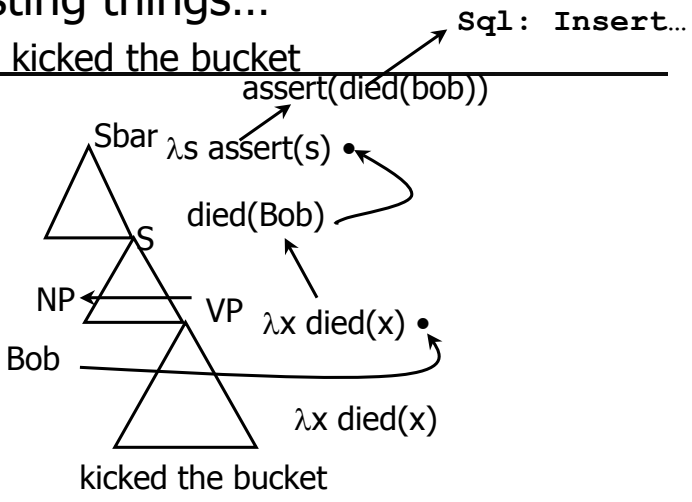
DB – the 'truth' as SQL (Datalog…)

- `Insert into Love(lover, loved) values(b, i)`
- Then we can answer a question, e.g.,
  Does Bob love ice-cream → via the SQL:

```
select 'yes' from Love where Love.lover = b
  and Love.loved = i
```


Now we can already do some interesting things…

- Bob kicked the bucket



And more

- I want a flight from NP [sem =x] to NP[sem =y]
- S [showflights(x, y)]
- **`Select from Flights`**…

Are we done?

- I wish!
- All the NPs so far are proper names, and so 'constants' – referring expressions
- Now we must consider lots more…
- The ice-cream, an ice-cream on the table, every ice-cream,…so much ice-cream, so little time…
- Bob likes no ice-cream…
- Plurals, tense, adverbs, adjectives, events,…

What about determiners?

- Bob loves an ice-cream
- This doesn't mean love(bob, ice-cream)
- Because just returns T or F – the same as
  - prime(17)
  - equal(4,2+2)
  - love(GeorgeWBush, LauraBush)
- What about "Bob loves an ice-cream and Sally loves an ice-cream"

The trouble with tribbles

- What would FOPC be for:
- Every person loves ice-cream, A person loves ice-cream

  $\forall$x (Person x $\rightarrow$love(x, ice-cream)
  $\exists$x (Person x & love(x, ice-cream)

Let's try our λ trick out on this…

Quantifiers cause problems

- If we apply composition following the syntax, what do we get?
- $\lambda y\ \lambda x$ love(x, y) ice-cream, $\forall x$ (Person x)
- But this yields:

love($\forall x$ (Person x), ice-cream)

NOT WHAT WE WANT:

$\forall x$ (Person x $\rightarrow$ love(x, ice-cream))

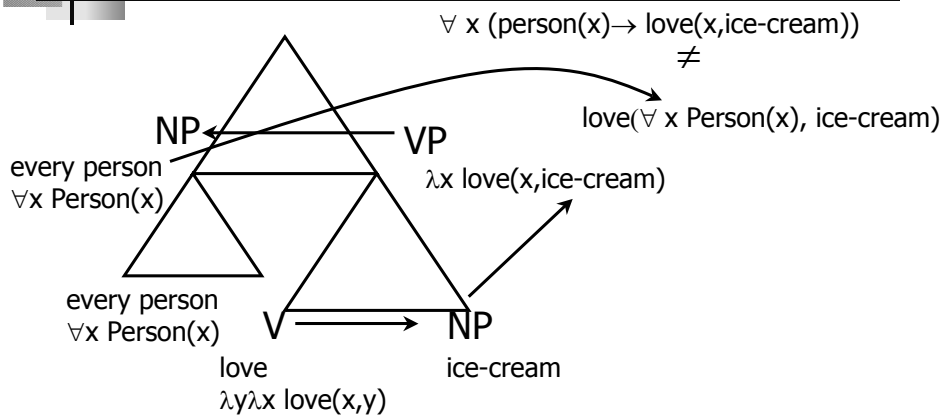
What happened to the NP 'every person'?

Direct substitution per syntactic isomorphism DOES NOT work

"every person" is not something like "Bob"

What to do???

With quantifiers, we lose direct substitution/isomorphism trick



This is the same problem as this…

- Nothing is better than a life of peace and prosperity
- John Kerry is better than nothing
- John Kerry is better than a life of peace and prosperity
- Bob sleeps = sleep(b); BUT
- No person sleeps = $\neg (\exists x)(person(x) \& sleep(x))$

And it gets worse…

- Not only does this violate syntactic/semantic isomorphism but
- How to extend to other phrases, like "most students love ice-cream"?
- We'll start by figuring out the resulting form we want, and then working backwards from there

Try 'most'

- Most students like ice-cream
- (1) Most(x) (student(x)) love (x, ice-cream)
- (2) Most(x) (student(x)) & love (x, ice-cream)
- Neither gives the correct truth conditions
  - The first is true: if for most things, if they are students, they like ice-cream
  - The second is true: most things are students and like ice-cream
- We really want to quantify over students and nothing else

A solution

- Intuitively: want to restrict the domain of the quantifier, via a type:
- Most x: student(x) [read: most x, s.t. student(x)]
- Then we have:
  (Most x : student(x)) love (x, ice-cream)
- Similarly, for 'everbody', 'somebody':
  (Every x: person(x)) love (x, ice-cream)
- Semantics for Most: we need – 'most A are B' or:
  $A \cap B > A - B$

That will give the proper truth conditions – the question is – how to assemble this from parts?

Working backwards by 'division'

- Given an output form, the result of a lambda application of a function to its argument, then what was the function (or argument)?
- Analogously: if 6 = 2 * x, what was x ?
- Example:

  Output: f(b)= love(b, ice-cream) What is f?

  Input: λx love(x, ice-cream).b

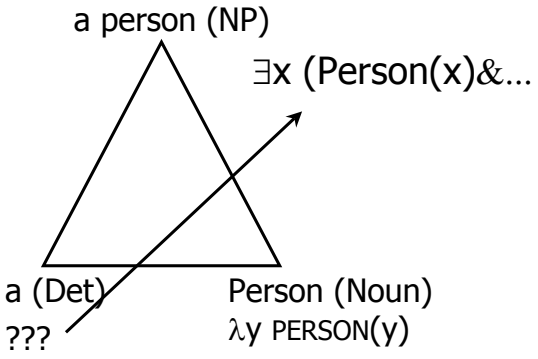
Simpler case: 'a person sleeps'

- Predicate: person (x)
- Lambda form: λx sleeps(x)
- Apply to bob: λx sleeps(x).bob & sleeps(bob)
- Now suppose the Noun has to be abstracted over <u>all</u> possible predicates (person, woman, child,…)

A person sleeps

- Output goal is this – think of it as a <u>template</u>
  $\exists x$ (Person(x) & sleeps(x))
- Let's do the Person(x) part – comes from 'a'…

NP 'a person' – to output:



Function ⟶ Argument (but it is also the name of a function!)

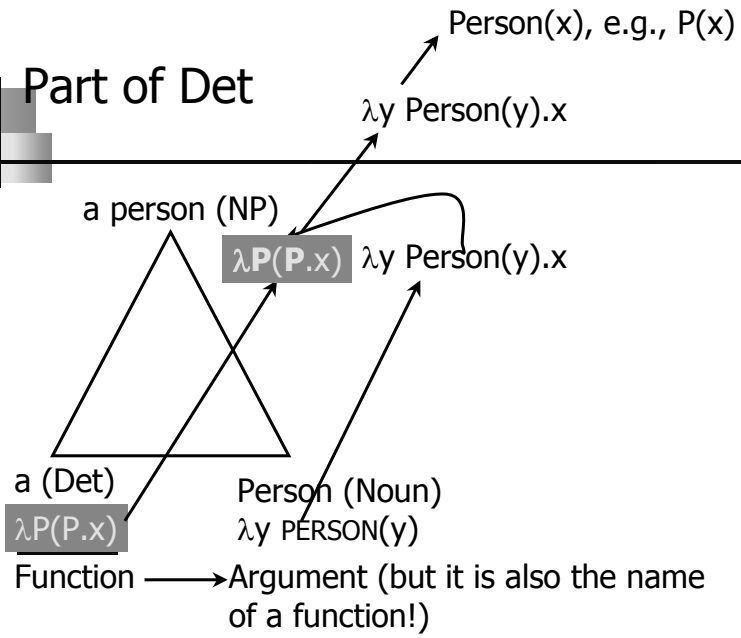
What is Det as a function?

- It acts as function applied to the arg (the lambda form) associated with the Noun
- It takes as input λy PERSON(y) and outputs

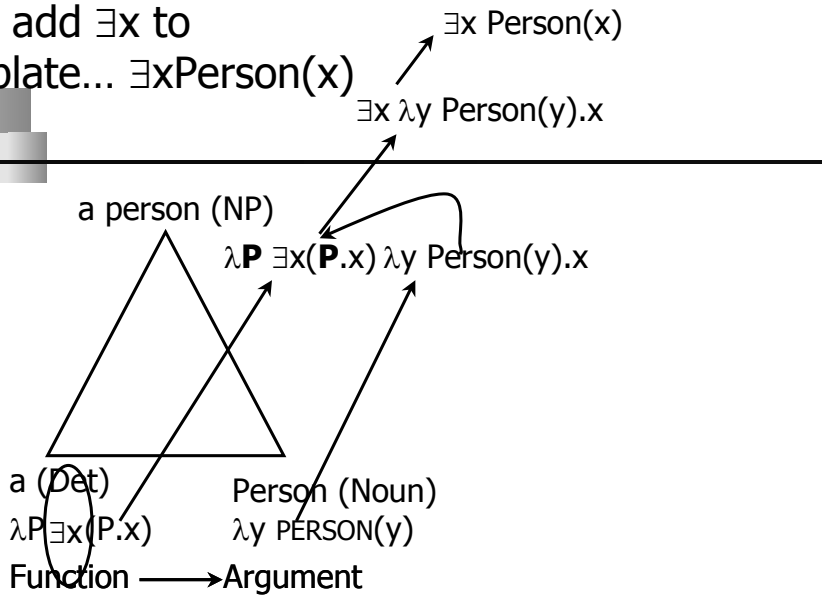
  λx PERSON(x)

- And more generally, takes as input a function (a lambda form) and outputs a lambda form
- So we need at least this:

Part of Det

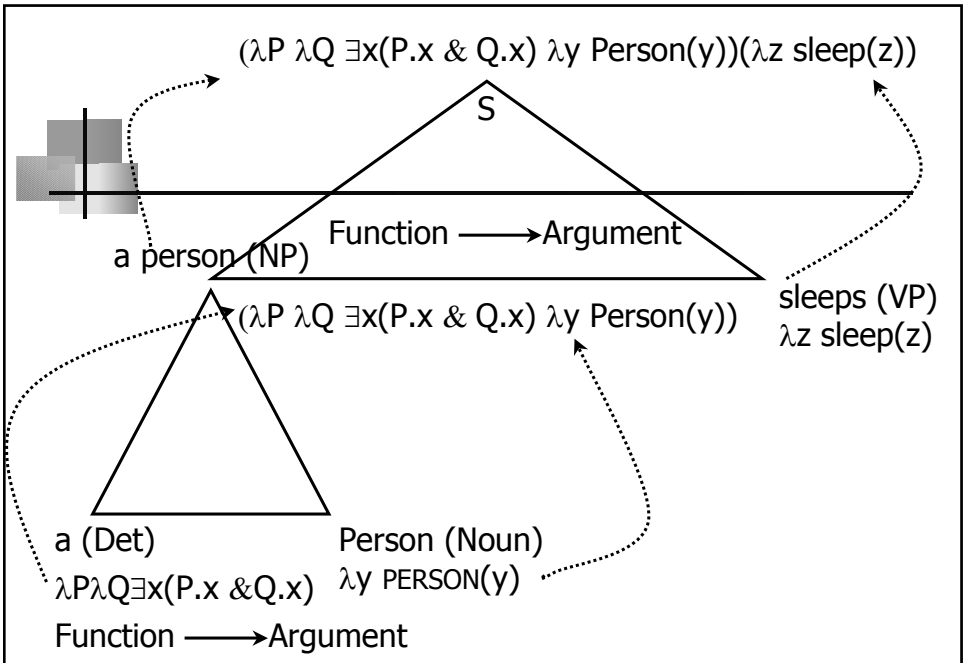


Now add ∃x to template… ∃xPerson(x)



Next incorporate intrans VP 'sleeps'

- As before, this is just λz sleep(z)
- Combines with NP, except now the NP is the function, and the VP is the argument
- Remember the output goal template is:
  ∃x (Person(x) & sleeps(x)) and in general,
  ∃x (P(x) & Q(x))
- And so far we have filled in:
  ∃x (Person(x)…
- So, Q(x) must be filled in by the other predicate, from the VP, Q(x)=λz sleep(z)



And after lambda substitution..

- We actually get what we want!

$\exists x$ (person(x) $\rightarrow$sleeps(x))

OK, what does this amount to?

We apply the Determiner as a function to the VP argument to get the sentence meaning

What happens to a name like "Bob"?

Now it must be a function- like "Everyone" …How can this be?

Bob sleeps = true iff the property 'sleeps' is among Bob's properties

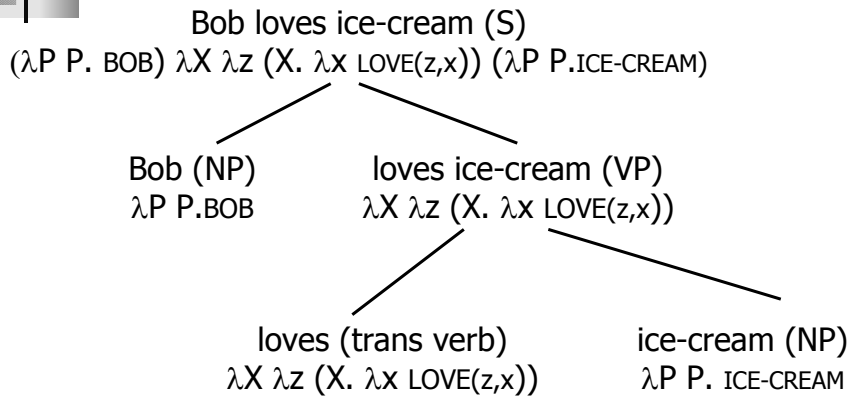
Names as functions

- Instead of "Bob" referring to an individual, it refers to the set of this individual's properties
- Each property is a set, so this is a set of sets
- λP(P.bob) = a set of sets
- This λ can now be applied as a function to any property as its argument e.g,
- λP P.bob .λx love(x, ice-cream) $\rightarrow$
  λx love(x, ice-cream).bob $\rightarrow$
  love(bob, ice-cream)
- So proper names and quantifiers treated uniformly
- Ice-cream = λP P.ice-cream

Derivation for "Bob loves ice-cream"

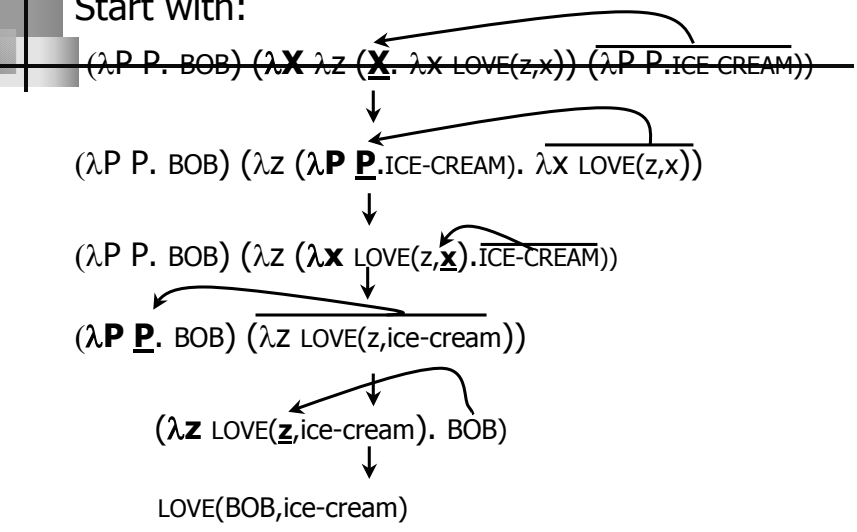
- Watch carefully – note parens
- The trick is to use the λ P's to get the predicates substituted
- We get a λ expression that is then 'reduced'
- Important: (Church-Rosser thm) – the order in which we do the applications to reduce the expression does not matter
- So, we can build up the template at the S however we like (from below, but in any order…)
- Plus important: this is MECHANICAL

The whole S – goal is love(bob, ice-cream)



Lambda reduction

Start with:



Now we can…uhm, go to town…

- Adjectives are properties

red: Adj: $\lambda P\ (\lambda x\ P(x) \wedge red'(x))$

- Prepositions are properties

in: P: $\lambda y\ \lambda P\ \lambda x\ (P(x) \wedge in'(y)(x))$

- …red Porsche in cambridge…

"…red porsche in cambridge"