An Introduction to Formal Computational Semantics

CS224N/Ling 280

Christopher Manning

May 23, 2000; revised 2008

A first example

Lexicon

Kathy, NP : **kathy**

Fong, NP : **fong**

respects, V : $\lambda y.\lambda x.\mathbf{respect}(x, y)$

runs, V : $\lambda x.\mathbf{run}(x)$

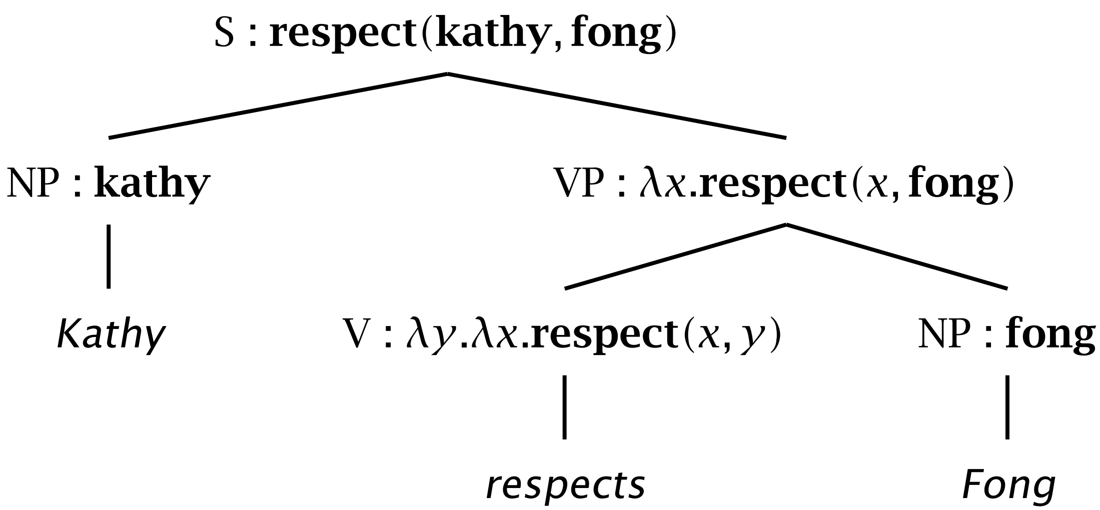
Grammar

S : $\beta(\alpha) \rightarrow$ NP : α VP : β

VP : $\beta(\alpha) \rightarrow$ V : β NP : α

VP : $\beta \rightarrow$ V : β

A first example

- 

- [$_{\text{VP}}$ respects Fong] : $[\lambda y.\lambda x.\mathbf{respect}(x, y)](\mathbf{fong})$

  $= \lambda x.\mathbf{respect}(x, \mathbf{fong})$ [β red.]

  [$_{\text{S}}$ Kathy respects Fong] : $[\lambda x.\mathbf{respect}(x, \mathbf{fong})](\mathbf{kathy})$

  $= \mathbf{respect(kathy,fong)}$

Database/knowledgebase interfaces

- Assume that **respect** is a table Respect with two fields respecter and respected
- Assume that **kathy** and **fong** are IDs in the database: k and f
- If we assert *Kathy respects Fong* we might evaluate the form **respect**(**fong**)(**kathy**) by doing an insert operation:

  insert into Respects(respecter, respected) values (k, f)

Database/knowledgebase interfaces

- Below we focus on questions like *Does Kathy respect Fong* for which we will use the relation to ask:

  select 'yes' from Respects where Respects.respecter = k and Respects.respected = f

- We interpret "no rows returned" as 'no' = **0**.

Typed λ calculus (Church 1940)

- Everything has a type (like Java!)
- **Bool** truth values (**0** and **1**)
  Ind individuals
  Ind → **Bool** properties
  Ind → **Ind** → **Bool** binary relations
- **kathy** and **fong** are **Ind**
  run is **Ind** → **Bool**
  respect is **Ind** → **Ind** → **Bool**
- Types are interpreted right associatively.
  respect is **Ind** → (**Ind** → **Bool**)
- We convert a several argument function into embedded unary functions. Referred to as *currying*.

Typed λ calculus (Church 1940)

- Once we have types, we don't need λ variables just to show what arguments something takes, and so we can introduce another operation of the λ calculus:

  η reduction [abstractions can be contracted]

  $\lambda x.(P(x)) \Rightarrow P$

- This means that instead of writing:

  $\lambda y.\lambda x.\mathbf{respect}(x, y)$

  we can just write:

  respect

Typed λ calculus (Church 1940)

- λ extraction allowed over any type (not just first-order)
- β reduction [application]

  $(\lambda x.P(\cdots, x, \cdots))(Z) \Rightarrow P(\cdots, Z, \cdots)$
- η reduction [abstractions can be contracted]

  $\lambda x.(P(x)) \Rightarrow P$
- α reduction [renaming of variables]

Typed λ calculus (Church 1940)

- The first form we introduced is called the β, η long form, and the second more compact representation (which we use quite a bit below) is called the β, η normal form. Here are some examples:
-

β, η normal form	β, η long form
run	$\lambda x.$**run**(x)
every²(kid, run)	**every²**$((\lambda x.$**kid**$(x))$, $(\lambda x.$**run**$(x))$
yesterday(run)	$\lambda y.$**yesterday**$(\lambda x.$**run**$(x))(y)$

Types of major syntactic categories

- nouns and verb phrases will be properties (**Ind** →**Bool**)
- noun phrases are **Ind** – though they are commonly type-raised to (**Ind** → **Bool**) → **Bool**
- adjectives are (**Ind** → **Bool**) → (**Ind** → **Bool**)

  This is because adjectives modify noun meanings, that is properties.
- Intensifiers modify adjectives: e.g, *very* in *a very happy camper*, so they're ((**Ind** → **Bool**) → (**Ind** → **Bool**)) → ((**Ind** → **Bool**) → (**Ind** → **Bool**)) [honest!].

A grammar fragment

- $S : \beta(\alpha) \to NP : \alpha \quad VP : \beta$
  $NP : \beta(\alpha) \to Det : \beta \quad N' : \alpha$
  $N' : \beta(\alpha) \to Adj : \beta \quad N' : \alpha$
  $N' : \beta(\alpha) \to N' : \alpha \quad PP : \beta$
  $N' : \beta \to N : \beta$
  $VP : \beta(\alpha) \to V : \beta \quad NP : \alpha$
  $VP : \beta(\gamma)(\alpha) \to V : \beta \quad NP : \alpha \quad NP : \gamma$
  $VP : \beta(\alpha) \to VP : \alpha \quad PP : \beta$
  $VP : \beta \to V : \beta$
  $PP : \beta(\alpha) \to P : \beta \quad NP : \alpha$

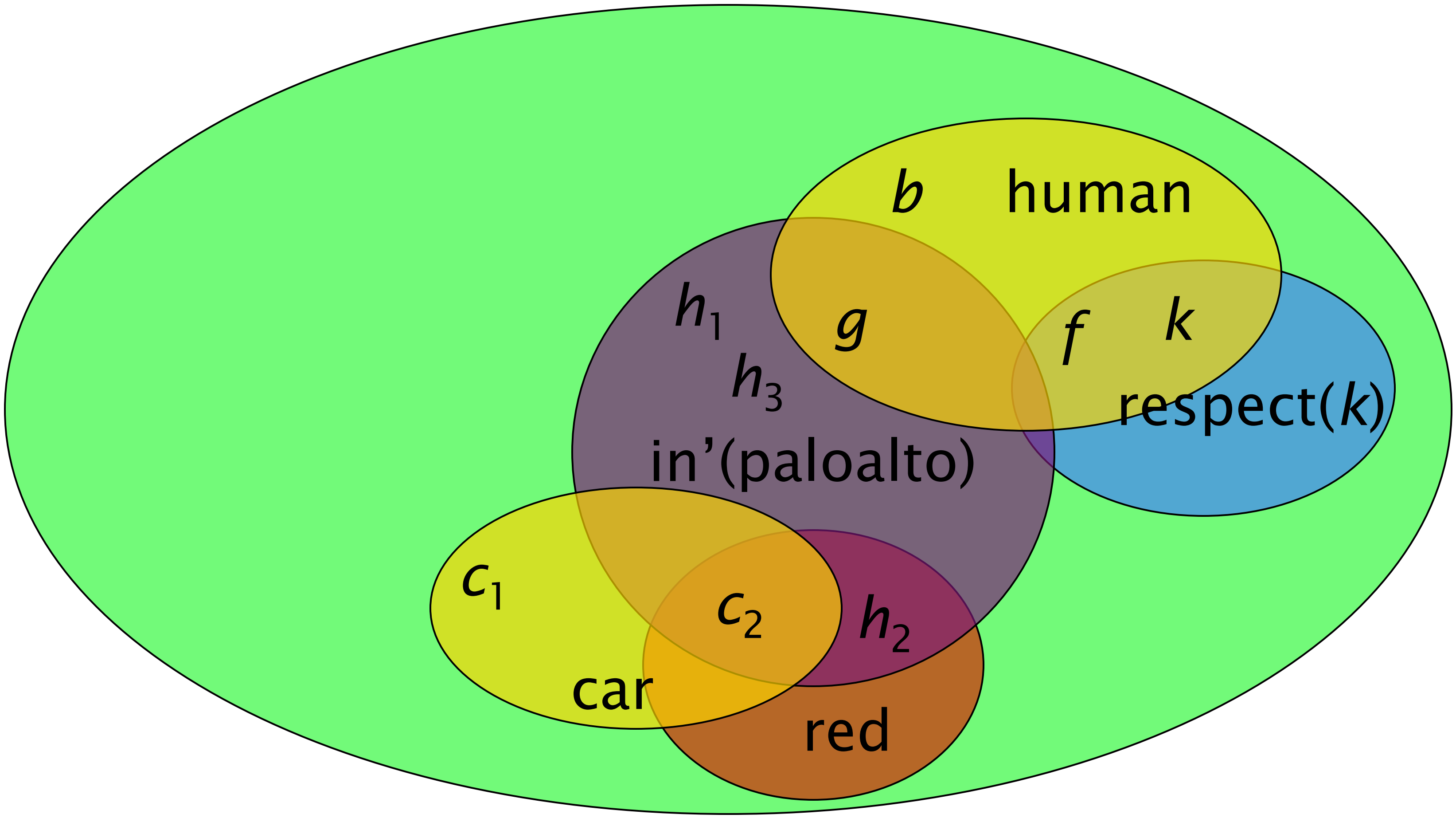
A grammar fragment

- *Kathy*, NP : $\mathbf{kathy}_{\mathbf{Ind}}$
  Fong, NP : $\mathbf{fong}_{\mathbf{Ind}}$
  Palo Alto, NP : $\mathbf{paloalto}_{\mathbf{Ind}}$
  car, N : $\mathbf{car}_{\mathbf{Ind} \to \mathbf{Bool}}$
  overpriced, Adj : $\mathbf{overpriced}_{(\mathbf{Ind} \to \mathbf{Bool}) \to (\mathbf{Ind} \to \mathbf{Bool})}$
  outside, PP : $\mathbf{outside}_{(\mathbf{Ind} \to \mathbf{Bool}) \to (\mathbf{Ind} \to \mathbf{Bool})}$
  red, Adj : $\lambda P.(\lambda x.P(x) \wedge \mathbf{red}'(x))$
  in, P : $\lambda y.\lambda P.\lambda x.(P(x) \wedge \mathbf{in}'(y)(x))$
  the, Det : ι
  a, Det : $\mathbf{some}^2_{(\mathbf{Ind} \to \mathbf{Bool}) \to (\mathbf{Ind} \to \mathbf{Bool}) \to \mathbf{Bool}}$
  runs, V : $\mathbf{run}_{\mathbf{Ind} \to \mathbf{Bool}}$
  respects, V : $\mathbf{respect}_{\mathbf{Ind} \to \mathbf{Ind} \to \mathbf{Bool}}$
  likes, V : $\mathbf{like}_{\mathbf{Ind} \to \mathbf{Ind} \to \mathbf{Bool}}$

A grammar fragment

- **in′** is **Ind** → **Ind** → **Bool**
- **in** $\stackrel{\text{def}}{=}$ $\lambda y.\lambda P.\lambda x.(P(x) \wedge$ **in′**$(y)(x))$ is **Ind** → (**Ind** → **Bool**) →(**Ind** → **Bool**)
- **red′** is **Ind** → **Bool**
- **red** $\stackrel{\text{def}}{=}$ $\lambda P.(\lambda x.(P(x) \wedge$ **red′**$(x))$ is (**Ind** → **Bool**) →(**Ind** → **Bool**)

Model theory – A formalization of a "database"



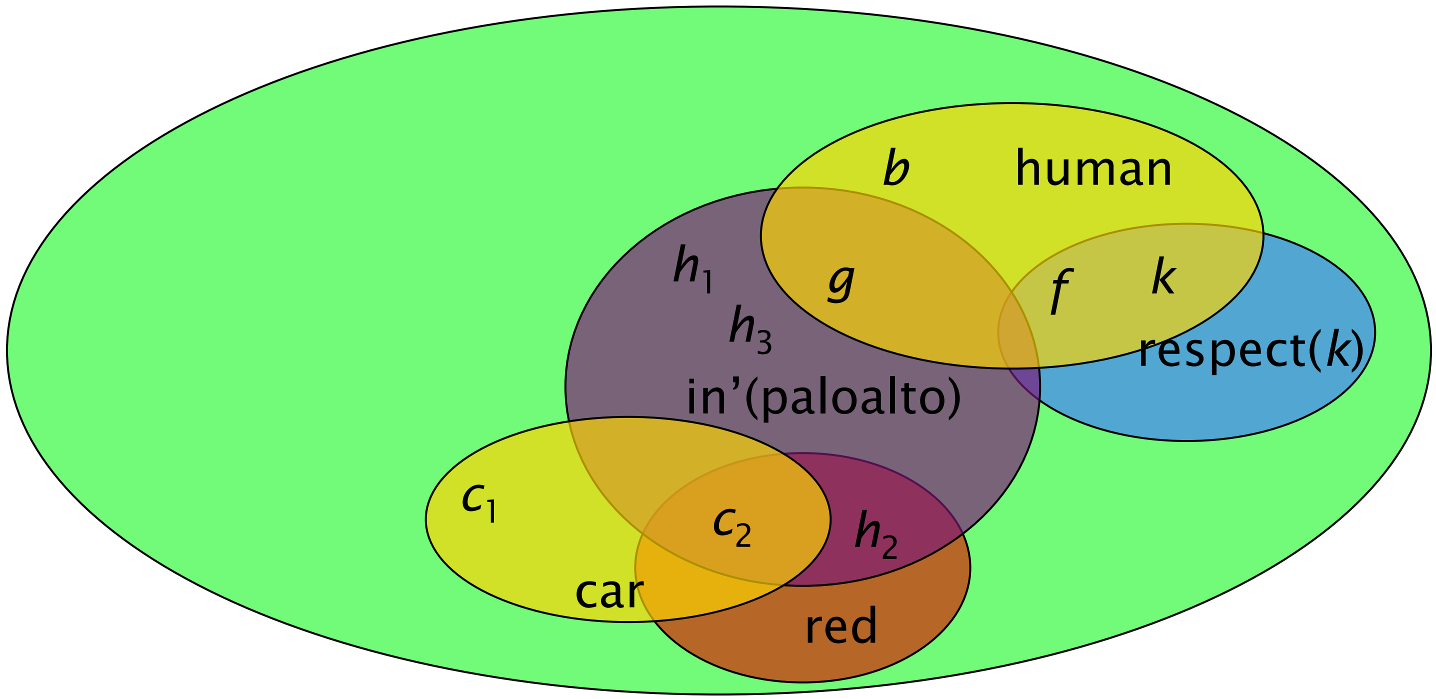
Properties

Curried multi-argument functions

$$[\![\textbf{respect}]\!] = [\![\lambda y.\lambda x.\textbf{respect}(x,y)]\!] = \left[\begin{array}{l} f \mapsto \begin{bmatrix} f \mapsto \mathbf{0} \\ k \mapsto \mathbf{1} \\ b \mapsto \mathbf{0} \end{bmatrix} \\ k \mapsto \begin{bmatrix} f \mapsto \mathbf{1} \\ k \mapsto \mathbf{1} \\ b \mapsto \mathbf{0} \end{bmatrix} \\ b \mapsto \begin{bmatrix} f \mapsto \mathbf{1} \\ k \mapsto \mathbf{0} \\ b \mapsto \mathbf{0} \end{bmatrix} \end{array}\right.$$

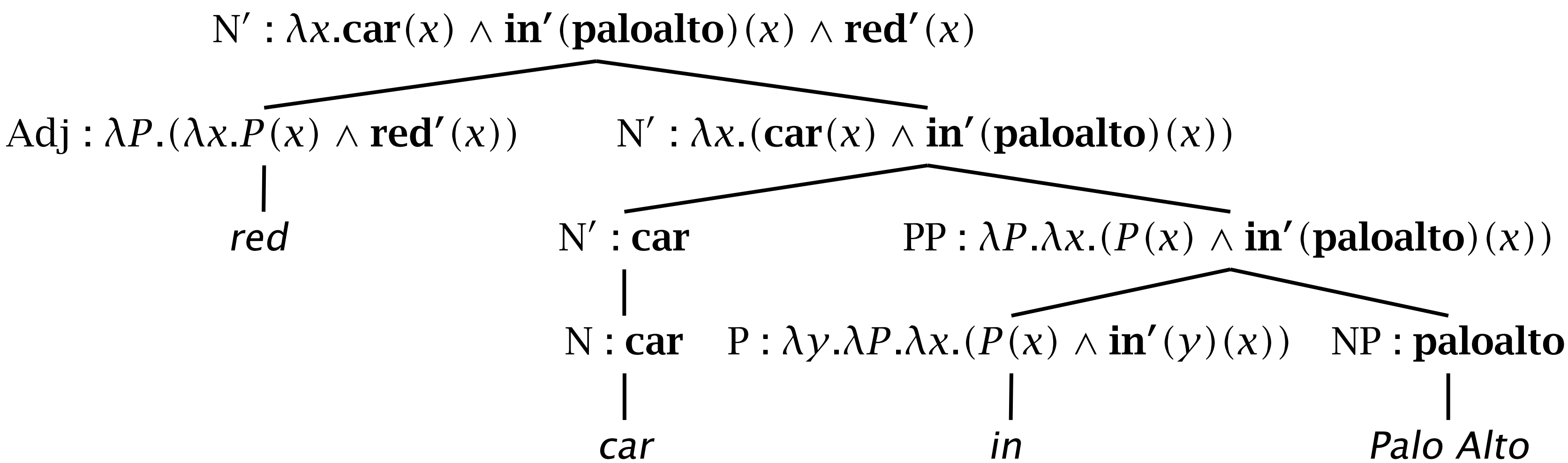
$$[\![\lambda x.\lambda y.\textbf{respect}(y)(x)(b)(f)]\!] = \mathbf{1}$$

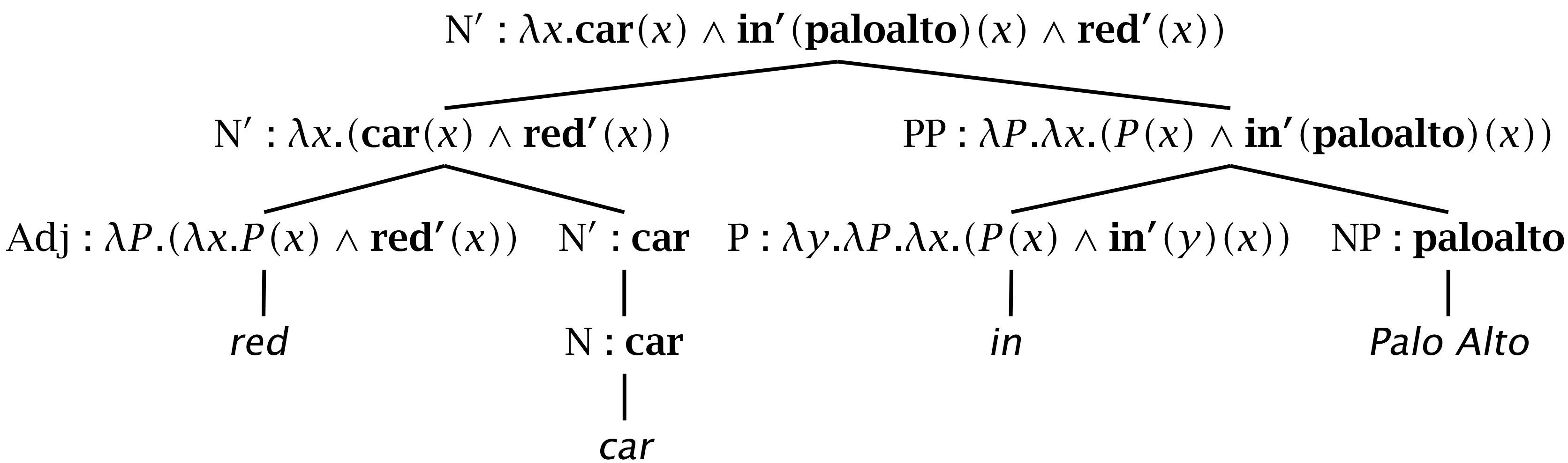
Quiz question



- Which individuals are the red things in Palo Alto?
- Who respects kathy (*k*)?

Adjective and PP modification

- 

- 

Intersective adjectives

- Syntactic ambiguity is spurious: you get the same semantics either way
- Database evaluation is possible via a table join

Non-intersective adjectives

- For non-intersective adjectives get different semantics depending on what they modify
- **overpriced**(**in**(**paloalto**)(**house**))
- **in**(**paloalto**)(**overpriced**(**house**))
- But probably won't be able to evaluate it on database!

Adding more complex NPs

NP: A man ~> $\exists x.man(x)$

S: A man loves Mary

~> * $love(\exists x.man(x), mary)$

- How to fix this?

A disappointment

Our first idea for NPs with determiner didn't work out:

"A man" ~> $\exists z.man(z)$

"A man loves Mary" ~> * $love(\exists z.man(z),mary)$

But what was the idea after all?

Nothing!

$\exists z.man(z)$ just isn't the meaning of "a man".

If anything, it translates the complete sentence

"There is a man"

Let's try again, systematically…

A solution for quantifiers

What we want is:

“A man loves Mary” ~> $\exists z(man(z) \wedge love(z,mary))$

What we have is:

“man” ~> $\lambda y.man(y)$

“loves Mary” ~> $\lambda x.love(x,mary)$

How about: $\exists z(\lambda y.man(y)(z) \wedge \lambda x.love(x,mary)(z))$

Remember: We can use variables for any kind of term.

So next:

$\lambda P(\lambda Q.\exists z(P(z) \wedge Q(z)))$ <~ “A”

Why things get more complex

- When doing predicate logic did you wonder why:
  - *Kathy runs* is $\textbf{run}(\textbf{kathy})$
  - *no kid runs* is $\neg(\exists x)(\textbf{kid}(x) \wedge \textbf{run}(x))$
- Somehow the NP's meaning is wrapped around the predicate
- Or consider why this argument doesn't hold:
  - Nothing is better than a life of peace and prosperity.
    A cold egg salad sandwich is better than nothing.
    ____
    A cold egg salad sandwich is better than a life of peace and prosperity.
- The problem is that *nothing* is a quantifier

Generalized Quantifiers

- We have a reasonable semantics for *red car in Palo Alto* as a property from **Ind** → **Bool**
- How do we represent noun phrases like *the red car in Palo Alto* or *every red car in Palo Alto*?
- $[\![\iota]\!](P) = a$ if ($P(b) = \mathbf{1}$ iff $b = a$)

  undefined, otherwise
- The semantics for *the* following Bertrand Russell, for whom *the* x meant the unique item satisfying a certain description

Generalized Quantifiers

- *red car in Palo Alto*

  select Cars.obj from Cars, Locations, Red where

  Cars.obj = Locations.obj AND

  Locations.place = ‘paloalto’ AND Cars.obj = Red.obj

  (here we assume the unary relations have one field, obj).

Generalized Quantifiers

- *the red car in Palo Alto*
- NP : $\iota(\lambda x.\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x) \wedge \mathbf{red'}(x))$

  Det : ι N′ : $\lambda x.\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x) \wedge \mathbf{red'}(x)$

  the *red car in Palo Alto*

- *the red car in Palo Alto*

  select Cars.obj from Cars, Locations, Red where

  Cars.obj = Locations.obj AND

  Locations.place = ‘paloalto’ AND Cars.obj = Red.obj

  having count(*) = 1

Generalized Quantifiers

- What then of *every red car in Palo Alto*?
- A generalized determiner is a relation between two properties, one contributed by the restriction from the N′, and one contributed by the predicate quantified over:

$$(\mathbf{Ind} \to \mathbf{Bool}) \to (\mathbf{Ind} \to \mathbf{Bool}) \to \mathbf{Bool}$$

- Here are some determiners

$$\mathbf{some}^2(\mathbf{kid})(\mathbf{run}) \equiv \mathbf{some}(\lambda x.\mathbf{kid}(x) \wedge \mathbf{run}(x))$$

$$\mathbf{every}^2(\mathbf{kid})(\mathbf{run}) \equiv \mathbf{every}(\lambda x.\mathbf{kid}(x) \to \mathbf{run}(x))$$

Generalized Quantifiers

- Generalized determiners are implemented via the quantifiers:

  $\textbf{every}(P) = \mathbf{1}$ iff $(\forall x)P(x) = \mathbf{1}$;

  i.e., if $P = \mathbf{Dom_{Ind}}$

  $\textbf{some}(P) = \mathbf{1}$ iff $(\exists x)P(x) = \mathbf{1}$; i.e., if $P \neq \emptyset$

Generalized Quantifiers

- Every student likes the red car
- $\mathrm{S} : \mathbf{every^2}(\mathbf{student})(\mathbf{like}(\iota(\lambda x.\mathbf{car}(x) \wedge \wedge \mathbf{red'}(x))))$
  - $\mathrm{NP} : \mathbf{every^2}(\mathbf{student})$
    - $\mathrm{Det} : \mathbf{every^2}$
      - *every*
    - $\mathrm{N'} : \mathbf{student}$
      - *student*
  - $\mathrm{VP} : \mathbf{like}(\iota(\lambda x.\mathbf{car}(x) \wedge \mathbf{red'}(x)))$
    - $\mathrm{V} : \mathbf{like}$
      - *likes*
    - $\mathrm{NP} : \iota(\lambda x.\mathbf{car}(x) \wedge \mathbf{red'}(x))$
      - $\mathrm{Det} : \iota$
        - *the*
      - $\mathrm{N'} : \lambda x.(\mathbf{car}(x) \wedge \mathbf{red'}(x))$
        - $\mathrm{Adj} : \lambda P.(\lambda x.P(x) \wedge \mathbf{red'}(x))$
          - *red*
        - $\mathrm{N'} : \mathbf{car}$
          - $\mathrm{N} : \mathbf{car}$
            - *car*

Representing proper nouns with quantifiers

- The central insight of Montague's PTQ was to treat individuals as of the same type as quantifiers (as type-raised individuals):
- *Kathy* : $\lambda P.P(\mathbf{kathy})$
- Both good and bad
- The main alternative (which we use) is flexible *type shifting* – you raise the type of something when necessary.

Nominal type shifting

- Common patterns of nominal type shifting
- $$\begin{tikzcd}
\mathbf{Ind} \arrow[rr, shift left, "\mathbf{Q}"] \arrow[dr, "\mathbf{R}"'] & & \mathbf{Ind} \to \mathbf{Bool} \arrow[ll, dashed, shift left, "\iota"] \arrow[dl, "\mathbf{some}^2"] \\
& (\mathbf{Ind} \to \mathbf{Bool}) \to \mathbf{Bool} &
\end{tikzcd}$$
- **R**(x) = $\lambda P.P(x)$
  some^2(P) = $\lambda Q.(Q \cap P) \neq \emptyset$
  Q(x) = $\lambda y.x = y$
- In this diagram, **R** is exactly this basic type-raising function for individuals.

Noun phrase scope – following Hendriks (1993)

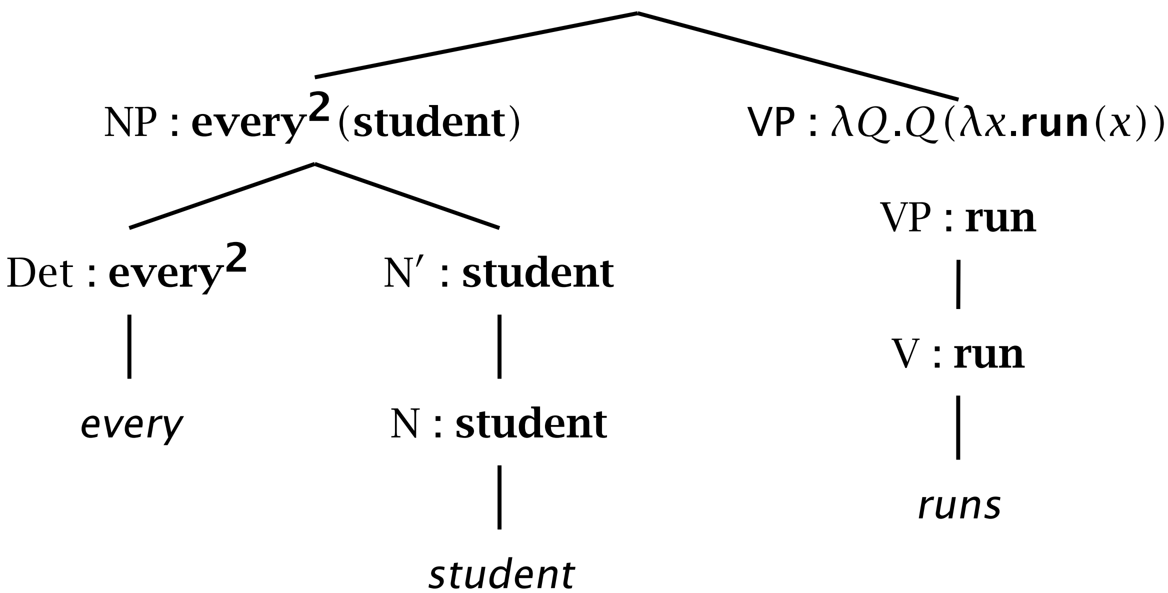
Value raising raises a function that produces an individual to one that produces a quantifer. If $\alpha : \sigma \to \mathbf{Ind}$ then $\lambda x.\lambda P.P(\alpha(x)) : \sigma \to (\mathbf{Ind}\to \mathbf{Bool})\to \mathbf{Bool}$

Argument raising replaces an argument of a boolean function with a variable and applies the quantifier semantically binding the replacing variable. If $\alpha : \sigma \to \mathbf{Ind} \to \tau \to \mathbf{Bool}$ then $\lambda x_1.\lambda Q.\lambda x_3.Q(\lambda x_2.\alpha(x_1)(x_2)(x_3)) : \sigma \to (\mathbf{Ind}\to \mathbf{Bool})\to \mathbf{Bool} \to \tau \to \mathbf{Bool}$

Argument lowering replaces a quantifier in a boolean function with an individual argument, where the semantics is calculated by applying the original function to the type raised argument. If $\alpha : \sigma \to ((\mathbf{Ind}\to \mathbf{Bool})\to \mathbf{Bool}) \to \tau \to \mathbf{Bool}$ then $\lambda x_1.\lambda x_2.\lambda x_3.\alpha(x_1)(\lambda P.P(x_2))(x_3) : \sigma \to \mathbf{Ind} \to \tau \to \mathbf{Bool}$

Every student runs

- $\mathrm{S} : \mathbf{every^2}(\mathbf{student})(\mathbf{run}) \equiv \mathbf{every}(\lambda x.\mathbf{student}(x) \to \mathbf{run}(x))$



Some kid broke every toy

- $\mathrm{S} : \mathbf{every^2}(\mathbf{toy})(\lambda y_o.\mathbf{some^2}(\mathbf{kid})(\lambda x_s.\mathbf{break}(y_o)(x_s)))$

$\mathrm{NP} : \mathbf{some^2}(\mathbf{kid})$

$\mathrm{VP} : \lambda S'.\mathbf{every^2}(\mathbf{toy})(\lambda y_o.S'(\lambda x_s.\mathbf{break}(y_o)(x_s)))$

$\mathrm{Det} : \mathbf{some^2}$

some

$\mathrm{N}' : \mathbf{kid}$

$\mathrm{N} : \mathbf{kid}$

kid

$\mathsf{V} : \lambda O.\lambda S'.O(\lambda y_o.S'(\lambda x_s.\mathbf{break}(y_o)(x_s)))$

$\mathsf{V} : \lambda x_o.\lambda S.S(\lambda x_s.\mathbf{break}(x_o)(x_s))$

$\mathrm{V} : \lambda y.\lambda x.\mathbf{break}(y)(x)$

broke

$\mathrm{NP} : \mathbf{every^2}(\mathbf{toy})$

$\mathrm{Det} : \mathbf{every^2}$

every

$\mathrm{N}' : \mathbf{toy}$

$\mathrm{N} : \mathbf{toy}$

toy

Some kid broke every toy

- $\mathrm{S} : \mathbf{some^2}(\mathbf{kid})(\lambda y_s.\mathbf{every^2}(\mathbf{toy})(\lambda x_o.\mathbf{break}(x_o)(y_s)))$

$\mathrm{NP} : \mathbf{some^2}(\mathbf{kid})$

$\mathrm{Det} : \mathbf{some^2}$ — *some*

$\mathrm{N'} : \mathbf{kid}$ — $\mathrm{N} : \mathbf{kid}$ — *kid*

$\mathrm{VP} : \lambda S.S(\lambda y_s.\mathbf{every^2}(\mathbf{toy})(\lambda x_o.\mathbf{break}(x_o)(y_s)))$

$\mathrm{V} : \lambda O'.\lambda S.S(\lambda y_s.O(\lambda x_o.\mathbf{break}(x_o)(y_s)))$

$\mathrm{V} : \lambda x_s.O(\lambda x_o.\mathbf{break}(x_o)(x_s))$

$\mathrm{V} : \lambda y.\lambda x.\mathbf{break}(y)(x)$

broke

$\mathrm{NP} : \mathbf{every^2}(\mathbf{toy})$

$\mathrm{Det} : \mathbf{every^2}$ — *every*

$\mathrm{N'} : \mathbf{toy}$ — $\mathrm{N} : \mathbf{toy}$ — *toy*

Questions with answers!

- A yes/no question (*Is Kathy running?*) will be something of type **Bool**, checked on database
- A content question (*Who likes Kathy?*) will be an *open proposition*, that is something semantically of the type *property* (**Ind** → **Bool**), and operationally we will consult the database to see what individuals will make the statement true.
- We use a grammar with a simple form of gap-threading for question words

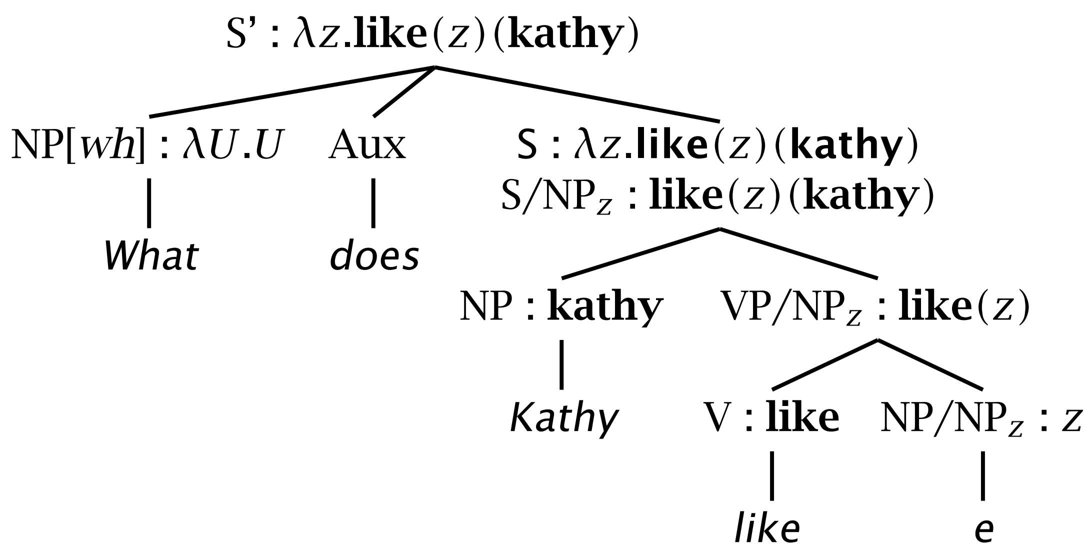
Syntax/semantics for questions

- $\mathsf{S}' : \beta(\alpha) \to \mathsf{NP}[wh] : \beta \quad \mathsf{Aux} \quad \mathsf{S} : \alpha$
  $\mathsf{S}' : \alpha \to \mathsf{Aux} \quad \mathsf{S} : \alpha$
  $\mathsf{NP/NP}_Z : z \to \mathsf{e}$
  $\mathsf{S} : \lambda z.F(\ldots z \ldots) \to \mathsf{S/NP}_Z : F(\ldots z \ldots)$

Syntax/semantics for questions

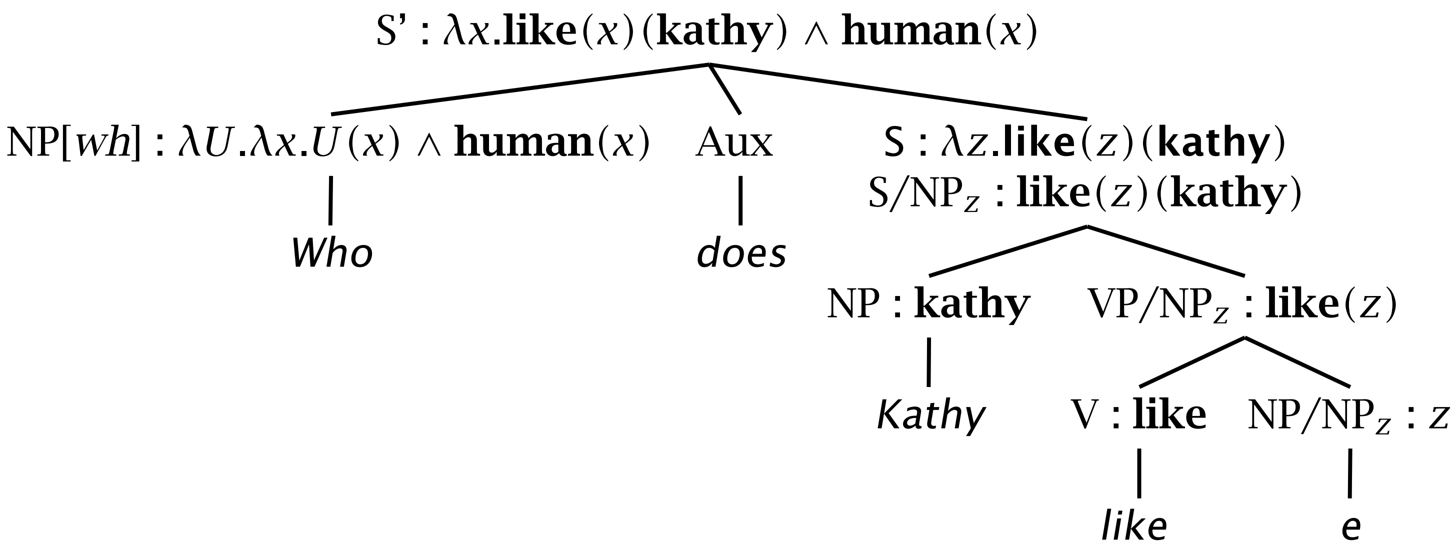
- *who*, NP[*wh*] : $\lambda U.\lambda x.U(x) \wedge \textbf{human}(x)$
  what, NP[*wh*] : $\lambda U.U$
  which, Det[*wh*] : $\lambda P.\lambda V.\lambda x.P(x) \wedge V(x)$
  how_many, Det[*wh*] : $\lambda P.\lambda V.|\lambda x.P(x) \wedge V(x)|$
- Where $|\cdot|$ is the operation that returns the cardinality of a set (count).

Question examples

- 

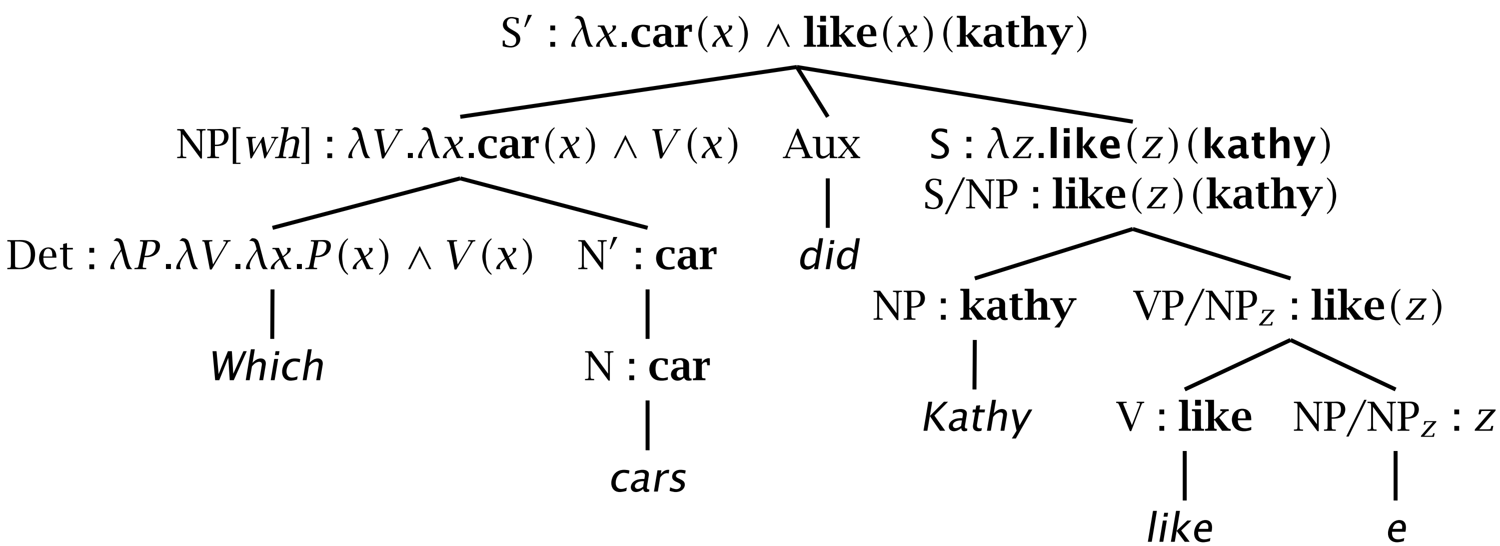
- select liked from Likes where Likes.liker='Kathy'

Question examples

- 

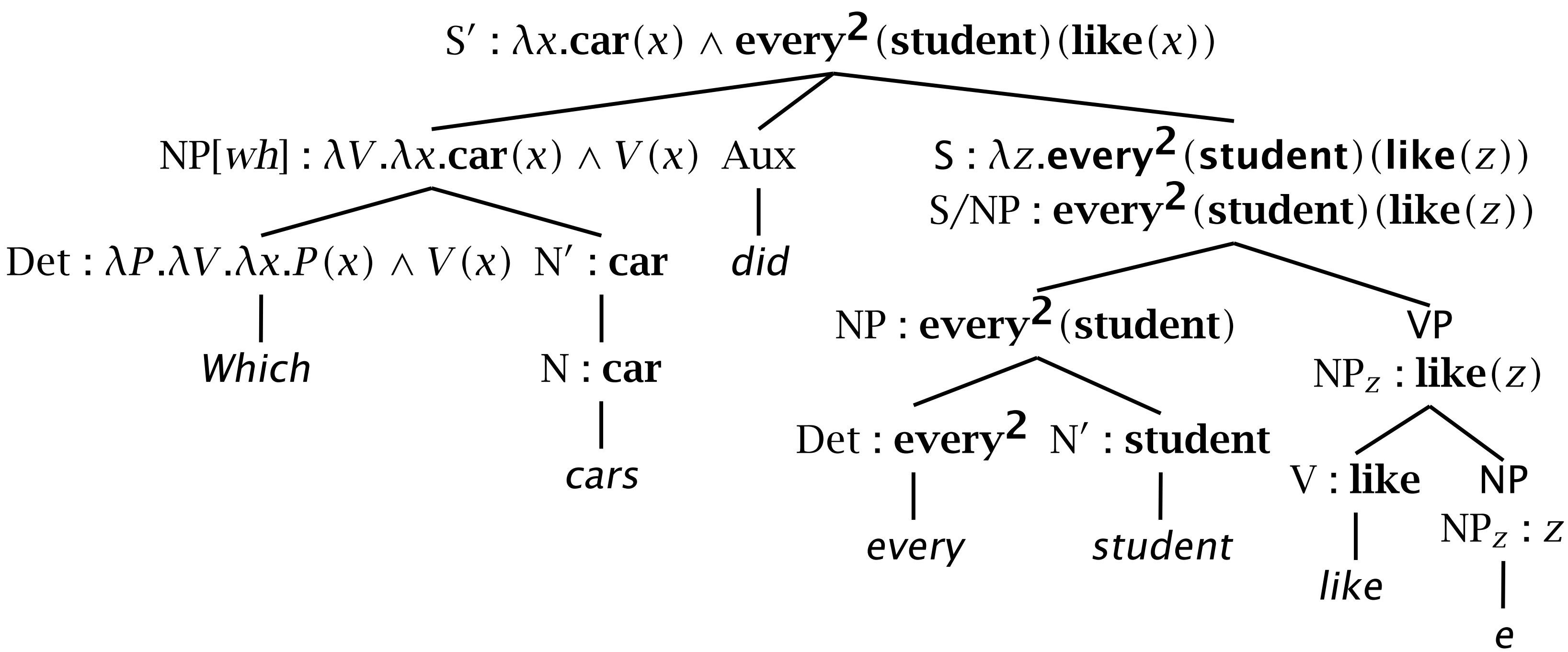
- select liked from Likes,Humans where Likes.liker='Kathy' AND Humans.obj = Likes.liked

Question examples

- 

- select liked from Cars,Likes where Cars.obj=Likes.liked AND Likes.liker='Kathy'

Question examples

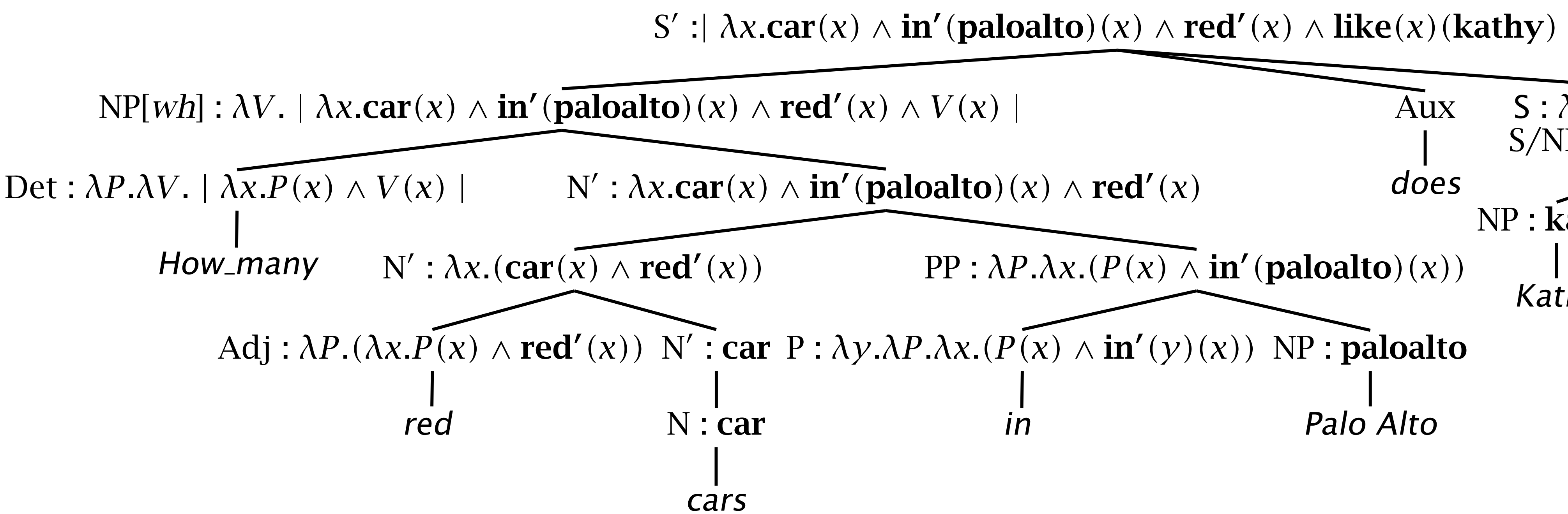
- 

- ???

Question examples

- *How many red cars in Palo Alto does Kathy like?*
- select count(*) from Likes,Cars,Locations,Reds where Cars.obj = Likes.liked AND Likes.liker = ’Kathy’ AND Red.obj = Likes.liked AND Locations.place = ’Palo Alto’ AND Locations.obj = Likes.liked
- *Did Kathy see the red car in Palo Alto?*
- select ‘yes’ where Seeings.seer = k AND Seeings.seen = (select Cars.obj from Cars, Locations, Red where Cars.obj = Locations.obj AND Locations.place = ‘paloalto’ AND Cars.obj = Red.obj having count(*) = 1)

How many red cars in Palo Alto does Kathy like?



Did Kathy see the red car in Palo Alto?

S' : $\mathbf{see}(\iota(\lambda x.\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x) \wedge \mathbf{red'}(x)))(\mathbf{kathy})$

Aux   S : $\mathbf{see}(\iota(\lambda x.\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x) \wedge \mathbf{red'}(x)))(\mathbf{kathy})$

Did   NP : $\mathbf{kathy}$ VP : $\mathbf{see}(\iota(\lambda x.\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x) \wedge \mathbf{red'}(x)))$

Kathy   V : $\mathbf{see}$ NP : $\iota(\lambda x.\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x) \wedge \mathbf{red'}(x))$

see   Det : ι N′ : $\lambda x.\mathbf{car}(x) \wedge \mathbf{in'}(\mathbf{paloalto})(x$

the   Adj : $\lambda P.(\lambda x.P(x) \wedge \mathbf{red'}(x))$ N′ : $\lambda x.(\mathbf{ca}$

red

N : $\mathbf{car}$

car   P : λ

How could we learn such representations?

- After disengagement for many years, there has started to be very interesting work in this area:
  - Luke S. Zettlemoyer and Michael Collins. 2005. Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars. In *Proceedings of the 21st UAI*.
  - Yuk Wah Wong and Raymond J. Mooney. 2007. Learning Synchronous Grammars for Semantic Parsing with Lambda Calculus. In *Proceedings of the 45th ACL*, pp. 960–967.

How could we learn such representations?

- General approach (ZC05): Start with initial lexicon, category templates, and paired sentences and meanings:

  What states border Texas?

  $\lambda x.\mathbf{state}(x) \wedge \mathbf{borders}(x, \mathbf{texas})$

- Learn lexical syntax/semantics for other words and learn to parse to logical form (parse structure is hidden).
- They successfully do iterative refinement of a lexicon and maxent parser

How can we reason with such representations?

- Logical reasoning is practical for certain domains (business rules, legal code, etc.) and has been used (see Blackburn and Bos 2005 for background).
- But our knowledge of the world is in general incomplete and uncertain.
- There is various recent work on handling *restricted* fragments of first order logic in probabilistic models
  - Lise Getoor, Nir Friedman, Daphne Koller, Avi Pfeffer, Benjamin Taskar. 2007. Probabilistic Relational Models. In *An Introduction to Statistical Relational Learning*. MIT Press.

How can we reason with such representations?

- Undirected model:
  - Pedro Domingos, Stanley Kok, Daniel Lowd, Hoifung Poon, Matthew Richardson, Parag Singla. 2008. Markov Logic. In L. De Raedt, P. Frasconi, K. Kersting and S. Muggleton (eds.), *Probabilistic Inductive Logic Programming*, pp. 92–117. Springer.
- A recent attempt to apply this to natural language inference:
  - Chloé Kiddon. 2008. Applying Markov Logic to the Task of Textual Entailment. Senior Honors Thesis, Computer Science. Stanford University.
- Logical formulae are given weights which are grounded out in an undirected markov network