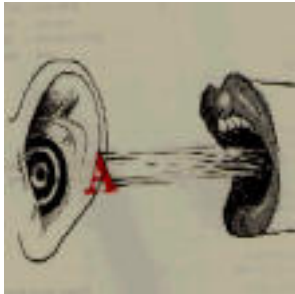


# Lecture 13: Semantics III



Professor Robert C. Berwick  
berwick@csail.mit.edu

# The Menu Bar

- Administrivia: Lab 4 due; Lab 5/6 out
- What does a more sophisticated lambda calculus to SQL mapping look like?
- Events, models, and determiners

# We'll do increasingly 'more challenging' sentences

1. Kathy respects Fong
2. Red car in Palo Alto
3. Overpriced house in Palo Alto
4. Kathy runs in Palo Alto
5. What does Kathy like
6. Who does Kathy like
7. Which cars did Kathy like
8. How many red cars in Palo Alto does Kathy like
9. Did Kathy see the red car in Palo Alto

# Paired syntactic-semantic rules

## 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 :  $\alpha$  VP :  $\beta$

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

VP :  $\beta \rightarrow$  V :  $\beta$

- Nonterminal : semantic translation translation rules say how to associate semantic representations with syntactic representations
- In general, head-nonhead syntactic composition corresponds to function application

# New lexicon

*Kathy*, NP : **kathy**<sub>Ind</sub>

*Fong*, NP : **fong**<sub>Ind</sub>

*Palo Alto*, NP : **paloalto**<sub>Ind</sub>

*car*, N : **car**<sub>Ind → Bool</sub>

*overpriced*, Adj : **overpriced**<sub>(Ind → Bool) → (Ind → Bool)</sub>

*outside*, PP : **outside**<sub>(Ind → Bool) → (Ind → Bool)</sub>

*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 :  $\iota$

*a*, Det : **some**<sup>2</sup><sub>(Ind → Bool) → (Ind → Bool) → Bool</sub>

*runs*, V : **run**<sub>Ind → Bool</sub>

*respects*, V : **respect**<sub>Ind → Ind → Bool</sub>

*likes*, V : **like**<sub>Ind → Ind → Bool</sub>

*sees*, V : **see**<sub>Ind → Ind → Bool</sub>

**in'** is **Ind → Ind → Bool**

**in**  $\cong \lambda y. \lambda P. \lambda x. (P(x) \wedge \mathbf{in}'(y)(x))$  is **Ind → (Ind → Bool) → (Ind → Bool)**

**red'** is **Ind → Bool**

**red**  $\cong \lambda P. (\lambda x. (P(x) \wedge \mathbf{red}'(x)))$  is **(Ind → Bool) → (Ind → Bool)**



## An aside about ‘types’

- You may have noticed that we associated ‘types’ to the objects in our Palo Alto universe - this is to make sure that functions apply to objects of the right sort
- There’s a standard notation for defining types in this sort of lambda calculus logic, starting with an inductive base
- There are two base types:
  1. ***Ind***, equivalently, ***e*** (for ‘individual’) is the ‘type’ of a basic entity, like *rocky*(but: we will alter this below!)
  2. ***Bool***, equivalently, ***t*** (for ‘truth value’) is the ‘type’ of a formula, which can be either true or false
- We can now form *complex types* for function expressions out of these; e.g.,  $\langle e, t \rangle$ , is the type of expressions that describe functions from entities to truth values, eg, unary predicates like *sleeps* or *runs*. (Eg., in our Palo Alto notation, we have  $run_{IND \rightarrow BOOL}$ )

# The complications

- What about *intransitive verbs*, e.g, ‘sleeps’?
- This is a function mapping to truth values, ie, *sleeps(x)* is true/false; so *sleeps(x)* is of type  $\langle e,t \rangle$

# More types

- What about *transitive* verbs?
- Remember, we can think of the VP as forming a verb of ‘1 argument’, once the object is found... so a VP, eg, *respects Fong* is just like an *intransitive verb*
- So a VP with a named object must be of the same type as a intransitive verb, namely,  $\langle e, t \rangle$
- And now the subject is of type  $e$ , so, the function mapping from Subject NPs to VPs must be of type:

$\langle t, \langle e, t \rangle \rangle$

This is why we have written  $respects_{IND \rightarrow IND \rightarrow BOOL}$   
(Putting aside for now the question of more complex subject types & the lambda calculus...)

## More types: names for things

- What about *names*?
- Recall: we change ‘rocky’ from being just a plain old constant, to be the ‘set of all predicates that are true of rocky’ - by using the lambda calculus
- What about types and the lambda calculus?
- We’ll say that if  $\alpha$  is of type  $\tau$  and  $x$  is a variable of type  $e$ , then  $\lambda x.\alpha$  is of type  $\langle e, \tau \rangle$
- So we can generalize (abstract over) *all* the predicates applicable to ‘rocky’ and say that ‘rock’ just denotes the set of all sets satisfying all the things true of rocky, ie,  
$$\lambda P.P(\text{rocky})$$
- This ‘type raising’ of names from  $\langle e \rangle$  to  $\langle e, t \rangle$  will turn out to match up well with what we need to describe logical determiners like ‘a’, ‘every’, etc.

## More about types: intransitive verbs

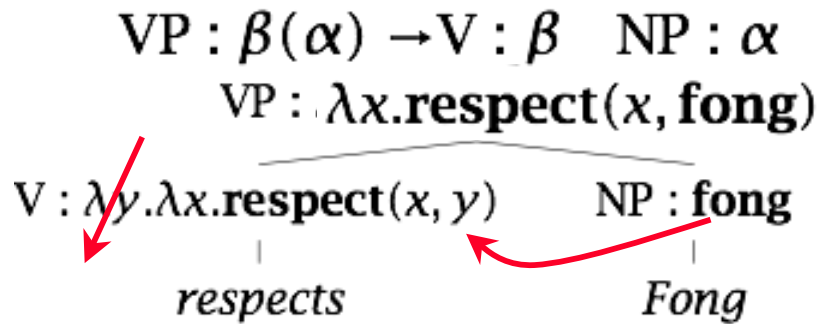
- So, if we have  $\lambda x.sleeps @rocky$ ; ('rocky' is of type  $e$  for the moment)
- Suppose we want to use one lambda abstract as the argument of *another* lambda abstract?
- We can't directly use  $\lambda y.y@rocky@(\lambda x.sleeps(x))$
- Why? Because  $\lambda y.y$  only applies to arguments of type  $e$  and the lambda form  $\lambda x.sleeps(x)$  is of type  $\langle e, t \rangle$  (as befits a function)
- Recall the hack: we need to abstract over types to get a *higher order* type - use the idea of a 'function over functions', since a unary function  $P$  is of type  $\langle e, t \rangle$ , we have that the whole abstract  $\lambda P.P$  is of type  $\langle \langle e, t \rangle, t \rangle$ , so we have:  
 $\lambda P.P@rocky@(\lambda x.sleeps(x))$ ; now we can substitute for  $P$  to get the form we want:  
 $\lambda x.sleeps(x)@rocky$  which reduces to  $sleeps(rocky)$
- All this shows is that our function applications are kosher

## More about types: transitive verbs

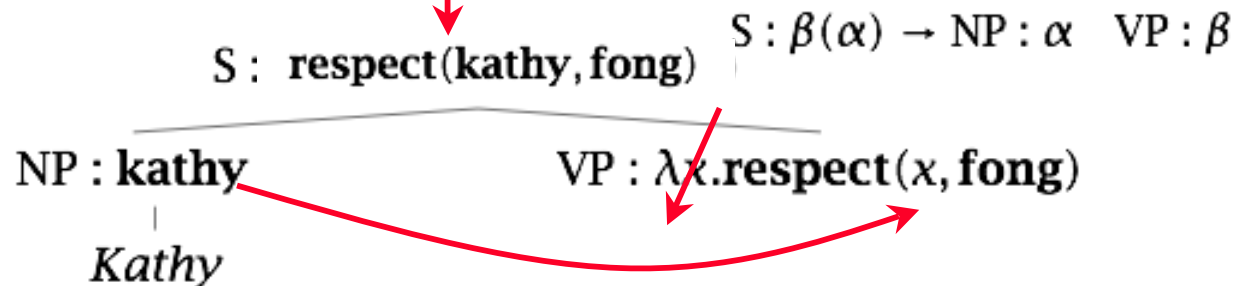
- The basic intuition: is that a verb of 2 arguments (the subject and the object), will behave like an intransitive verb (a verb of 1 argument), as soon as we fill in the object
- So, this means that complete VPs (with the object NP filled in) really should have the same type as an ordinary intransitive verb, ie,  $\langle e, t \rangle$  or

IND $\rightarrow$ BOOL

# Sequence of function applications (beta reductions) tells us how to compose the semantic representation



$$[\text{VP respects Fong}] : [\lambda y. \lambda x. \text{respect}(x, y)](\text{fong}) = \lambda x. \text{respect}(x, \text{fong}) \quad [\beta \text{ red.}]$$



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

insert into **Respects**(respector, respected) values (*k*, *f*)

## Abbreviate some notation...

$\lambda x.(P(x)) \Rightarrow P$   $\eta$  reduction [abstractions can be contracted]

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

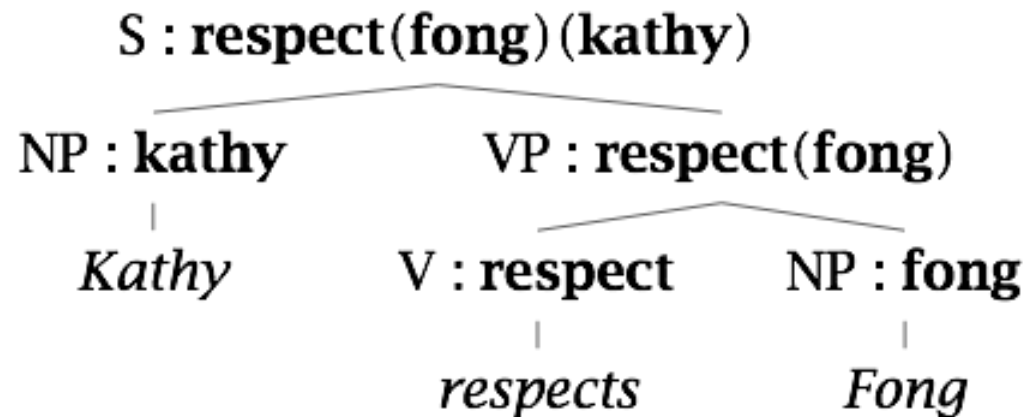
  
**respect**

**respect** is  $\text{Ind} \rightarrow \text{Ind} \rightarrow \text{Bool}$

$\beta, \eta$  long form:  $\lambda x.\mathbf{run}(x), \lambda y.\mathbf{yesterday}(\lambda x.\mathbf{run}(x))(y)$

$\beta, \eta$  normal form: **run**, **yesterday(run)**

So we can abbreviate the whole sentence semantics this way...



Which leads to an immediate SQL treatment:

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

Ok, what's next???

What about prepositions, adjectives, adverbs,...

# New, improved grammar, with semantic augmentation

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

# New lexicon

*Kathy*, NP : **kathy**<sub>Ind</sub>

*Fong*, NP : **fong**<sub>Ind</sub>

*Palo Alto*, NP : **paloalto**<sub>Ind</sub>

*car*, N : **car**<sub>Ind → Bool</sub>

*overpriced*, Adj : **overpriced**<sub>(Ind → Bool) → (Ind → Bool)</sub>

*outside*, PP : **outside**<sub>(Ind → Bool) → (Ind → Bool)</sub>

*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 :  $\iota$

*a*, Det : **some**<sup>2</sup><sub>(Ind → Bool) → (Ind → Bool) → Bool</sub>

*runs*, V : **run**<sub>Ind → Bool</sub>

*respects*, V : **respect**<sub>Ind → Ind → Bool</sub>

*likes*, V : **like**<sub>Ind → Ind → Bool</sub>

*sees*, V : **see**<sub>Ind → Ind → Bool</sub>

**in'** is **Ind → Ind → Bool**

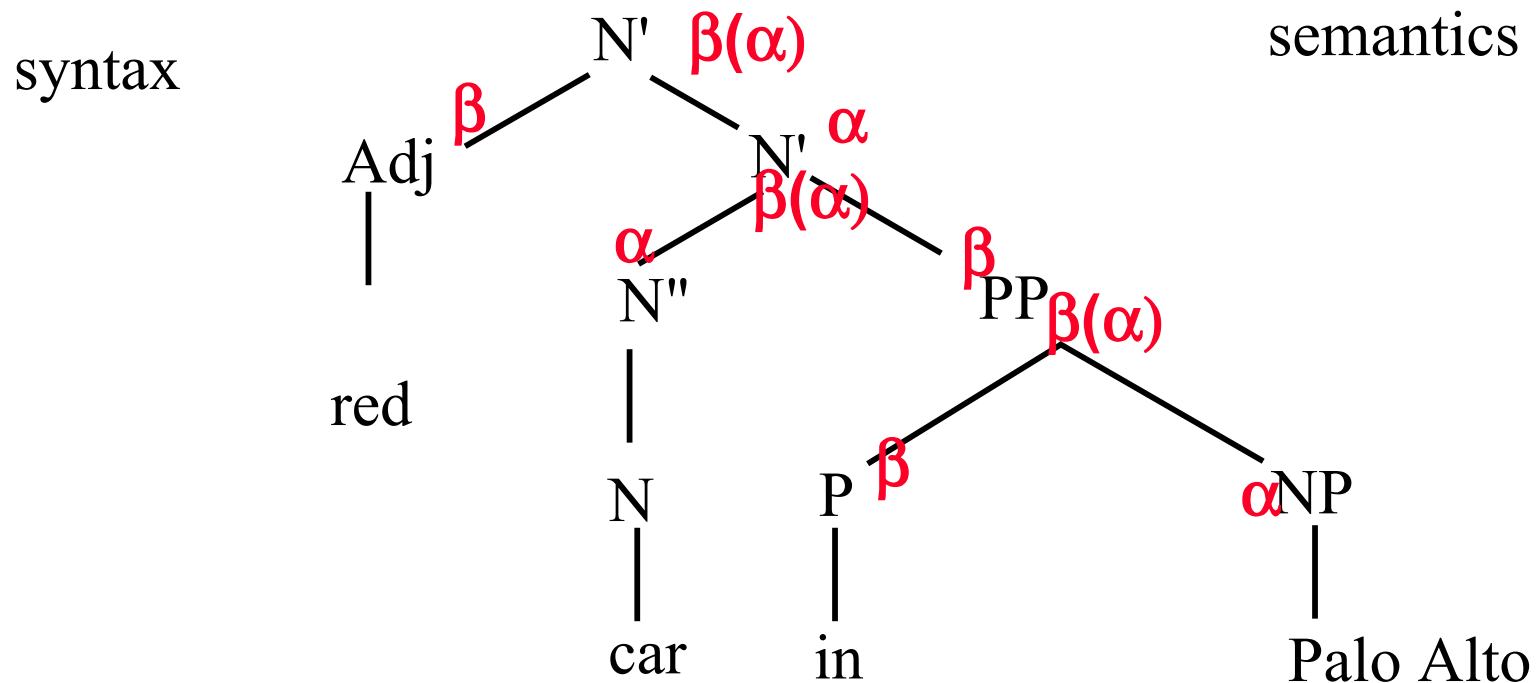
**in**  $\cong \lambda y. \lambda P. \lambda x. (P(x) \wedge \mathbf{in}'(y)(x))$  is **Ind → (Ind → Bool) → (Ind → Bool)**

**red'** is **Ind → Bool**

**red**  $\cong \lambda P. (\lambda x. (P(x) \wedge \mathbf{red}'(x)))$  is **(Ind → Bool) → (Ind → Bool)**

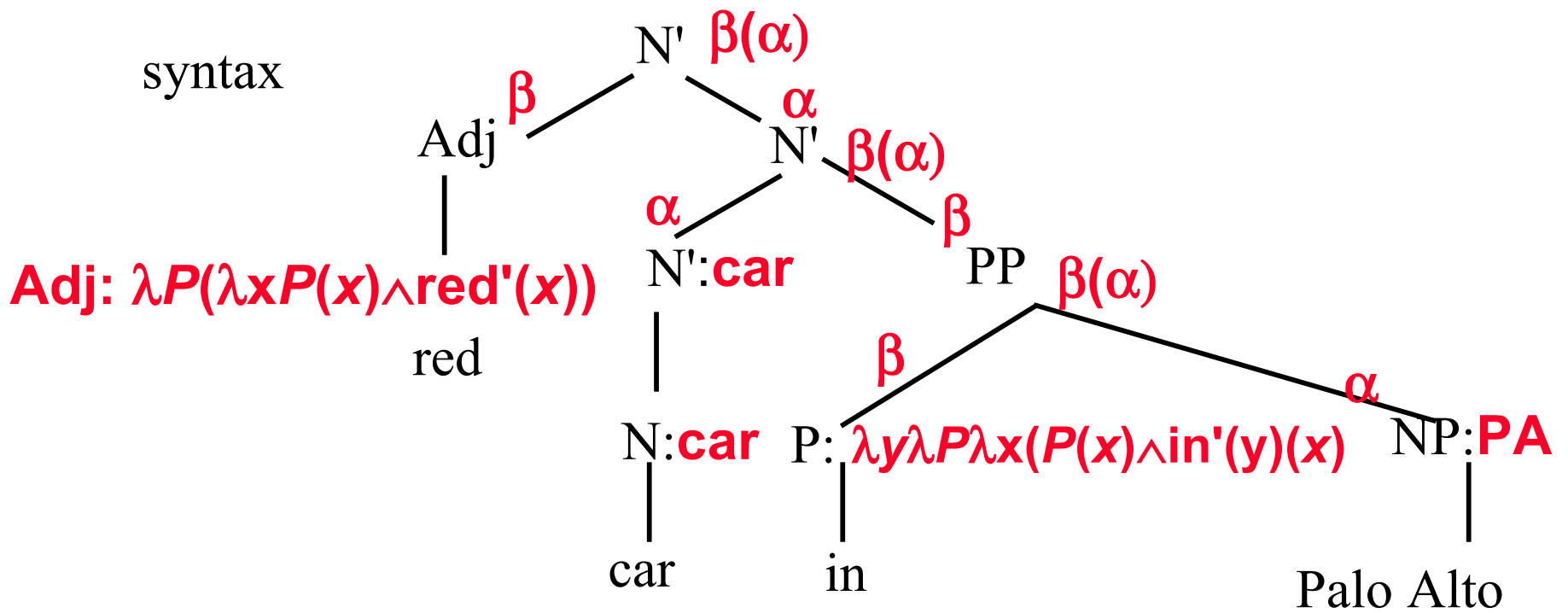
# Let's try it out

Desired Goal:  $\lambda x.\text{car}(x) \wedge \text{in}'(\text{paloalto})(x) \wedge \text{red}'(x)$   
*red car in Palo Alto*



Now add lexical entries and start doing function application...

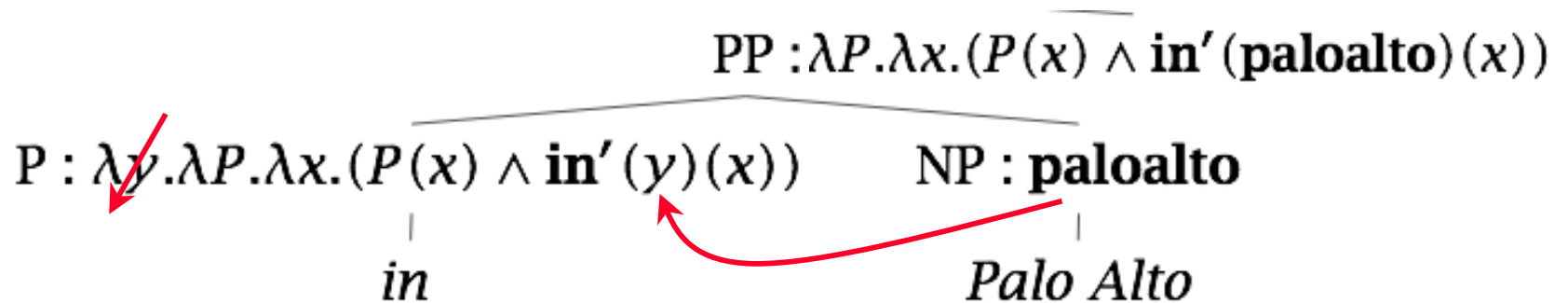
# Adding lexical entries



Now do function applications (beta reductions)

# Beta reduction 1: PP 'in Palo Alto'

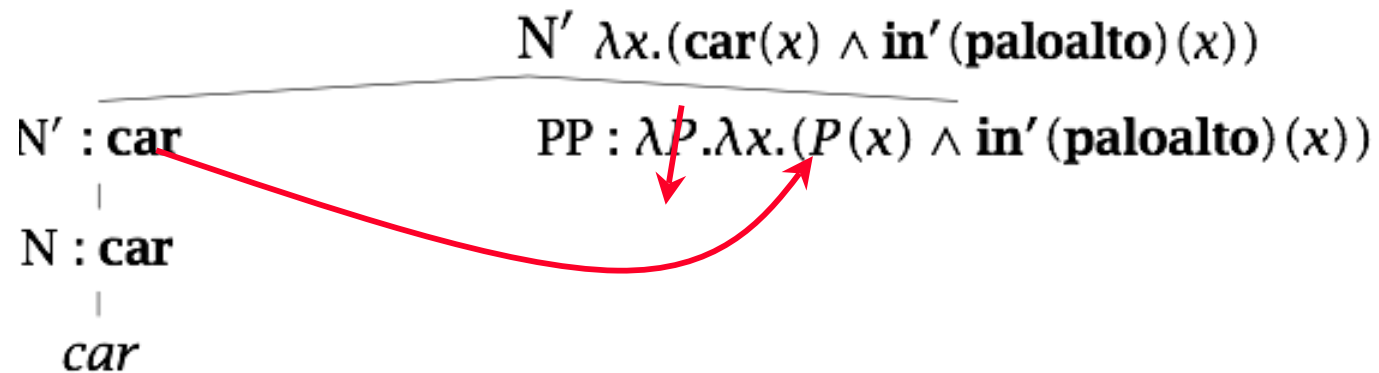
$$PP : \beta(\alpha) \rightarrow P : \beta \quad NP : \alpha$$



NB: Church-Rosser theorem assures us we could do these reductions *in any order* and get same result at the end – good for computation

# $\beta$ -reduction 2: ‘*car* [in Palo Alto]’

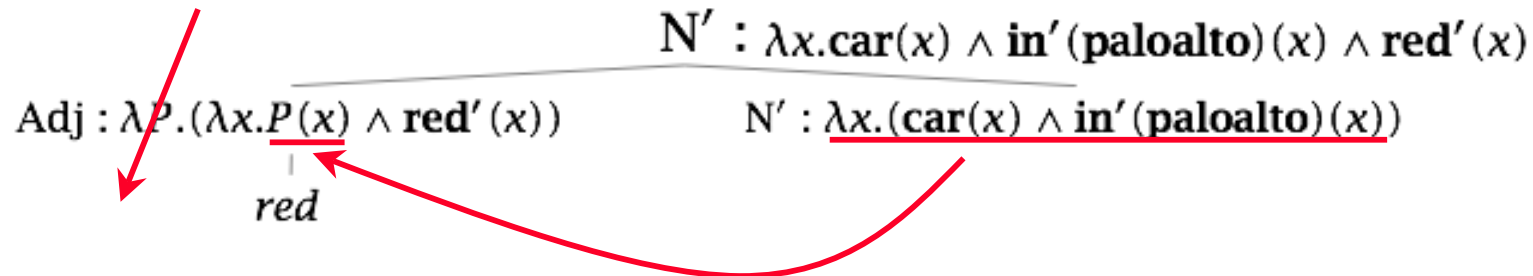
$$N' : \beta(\alpha) \rightarrow N' : \alpha \quad PP : \beta$$



$$N' : \beta \rightarrow N : \beta$$

# $\beta$ -reduction 3: ‘*red* [car in Palo Alto]’

$$N' : \beta(\alpha) \rightarrow \text{Adj} : \beta \quad N' : \alpha$$



$\text{red}, \text{Adj} : \lambda P. (\lambda x. P(\bar{x}) \wedge \text{red}'(\bar{x}))$

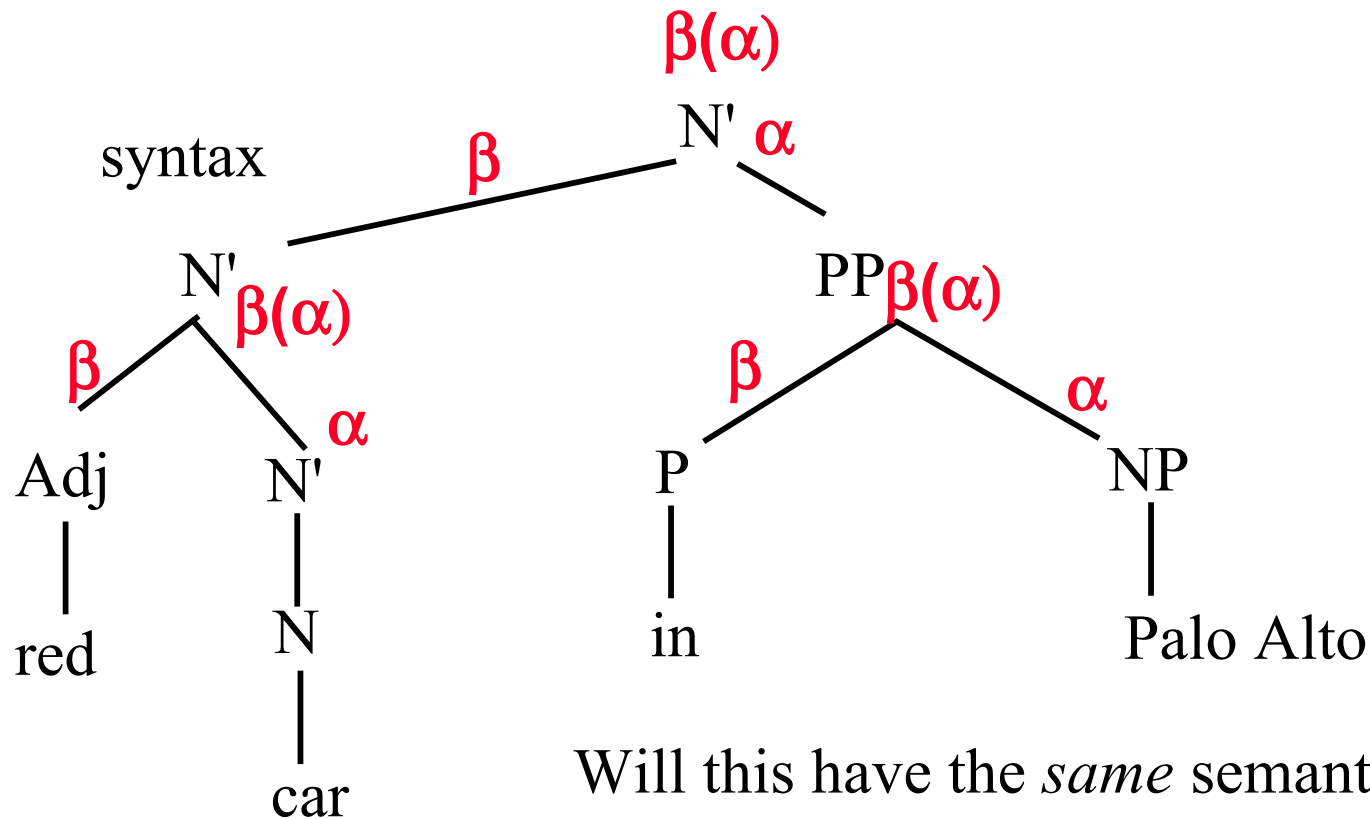
lexicon

Hah, it works!!

We're done!!! Well, almost...



## Parse #2 - what are *its* semantics?

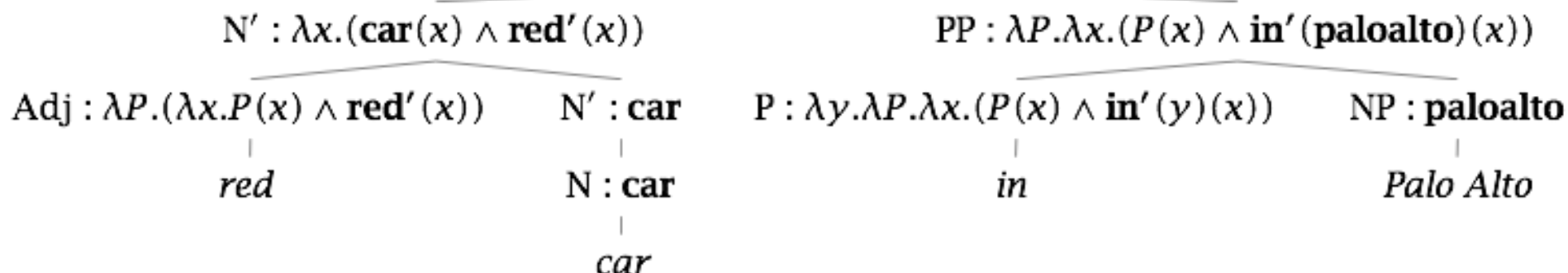


Will this have the *same* semantics?

What does  $\beta$ -reduction construct?

# After the beta dust settles...

$N' : \lambda x. \text{car}(x) \wedge \text{in}'(\text{paloalto})(x) \wedge \text{red}'(x)$



It's the same semantics as the first parse!!... why???

Because: **red** is an intersective adjective

(there are other adjectives that are non-intersective)

## More on adjectives

- Intersective:

Phrases like [blue suit] are easy to understand because we can intersect the set of blue things with the set of suits and get the set of blue suits. Such adjectives are called intersective adjectives

## Most typical kind of adjective

- Relative Intersection:
- [big] in [big mosquito] work differently
- what we mean is “big for an X”, in this case, “big for a mosquito”
- A big elephant and a big mouse are two very different senses of *big*, unless we understand *big* as an adjective whose meaning is relativized to the noun or noun phrase that it’s modifying
- This is called relative intersection, and *big* is an adjective of relative intersection. Other examples are *tall*, *good*, *short*, *poor*, *rich*

## And.....

- Non-intersection: Non-intersective adjectives are adjectives that don't entail reference to the objects denoted by the noun.
- Adjectives like *alleged* and *possible*
- If we say, *the alleged thief arrived in court*, we need not be talking about a thief (cf, *big mosquito*)

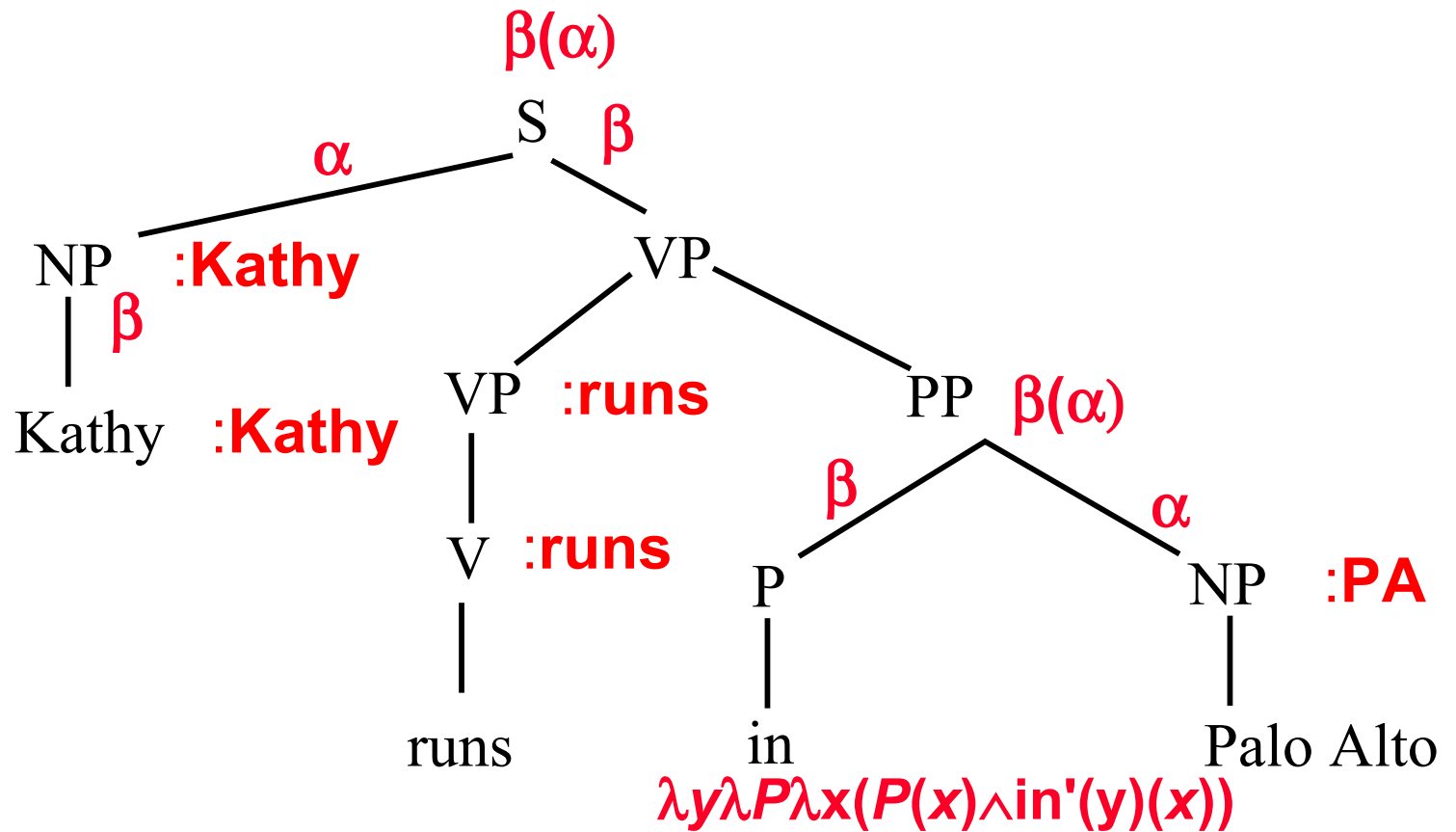
# Finally

- Anti-intersection: Anti-intersective adjectives are adjectives that can't entail reference to the objects denoted by the noun
- They entail non-reference to the noun in question
- This is best understood if we think about the phrase [*a fake Picasso*]
- The semantics of [*fake*] are such that the phrase [*a fake Picasso*] can never refer to a real painting by Picasso

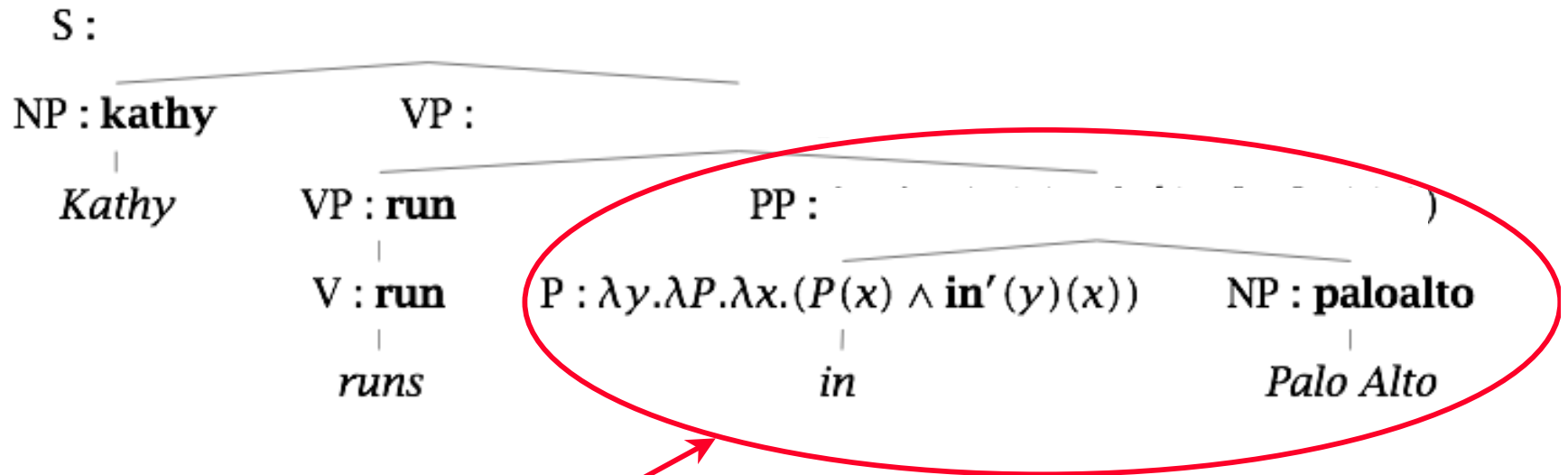
OK, now on to the marathon challenge...

- Kathy runs in Palo Alto
- A PP modifying a VP
- Note that a VP is of type (**Ind**  $\rightarrow$  **Bool**), that maps Individuals to Booleans (truth values), i.e., a *property*, just like an Name, so a PP can modify either...

# The syntax tree, decorated



# Now strut our lambda stuff...



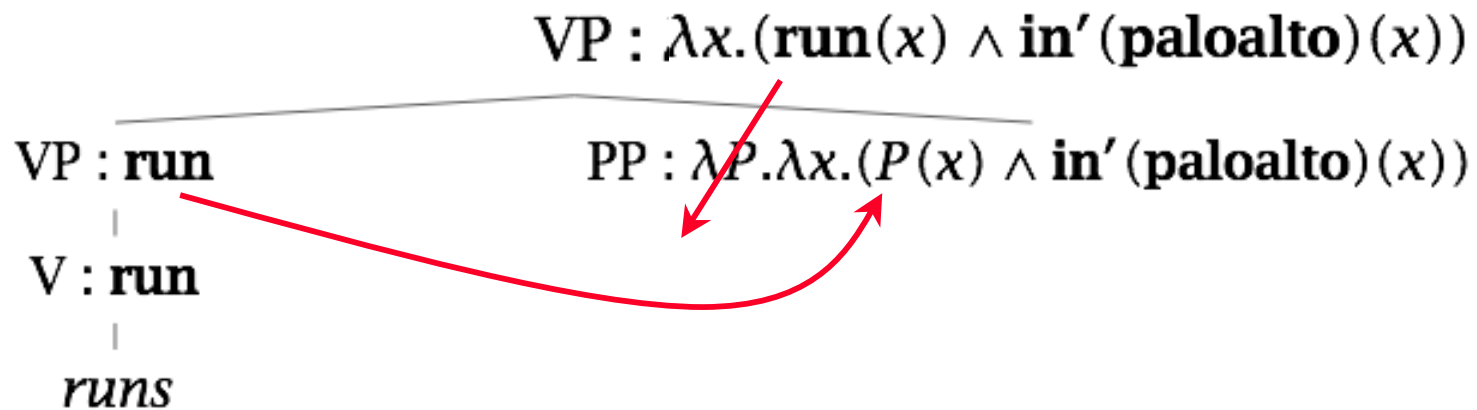
We already know the semantics for this PP...

$$\lambda P. \lambda x. (P(x) \wedge \mathbf{in}'(\mathbf{paloalto})(x))$$

So let's move on to the VP-PP...

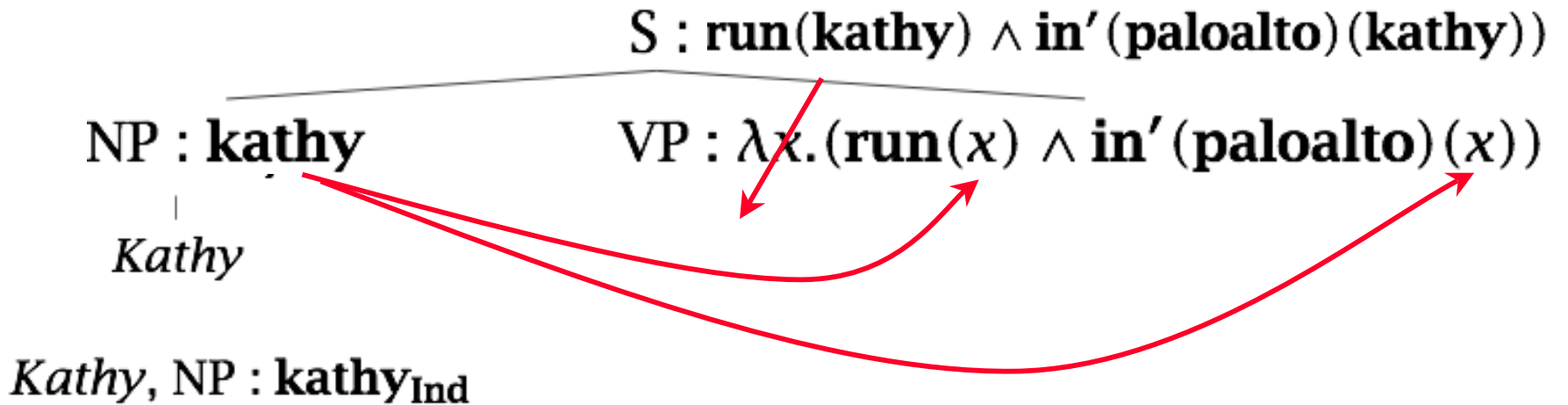
$VP: \beta \rightarrow V: \alpha \quad PP: \beta$

$VP: \beta \rightarrow V: \beta$



Now onto the NP-VP combination...

$S : \beta(\alpha) \rightarrow NP : \alpha \quad VP : \beta$



## “the”

‘the red car in Palo Alto’

- What does ‘the’ mean? We had it as “ι”
- Philosopher Russell: *the x* means ‘the unique item satisfying a certain description,’ or:
  - $[[\iota]](P) = a$  if  $(P(a) = 1 \text{ iff } b=a)$
- It is thus only a partial function, returning undefined if there is no such object (technically different from a false statement)
- So putting this into our parse, then SQL, we have:





# Questions with answers!

We need slashes; also need semantics for fillers and gaps...

$$S' : \beta(\alpha) \rightarrow \text{NP}[wh] : \beta \quad \text{Aux} \quad S : \alpha$$
$$S' : \alpha \rightarrow \text{Aux} \quad S : \alpha$$
$$\text{NP}/\text{NP}_z : z \rightarrow e$$

Gap introduces a variable

$$S : \lambda z.F(\dots z \dots) \rightarrow S/\text{NP}_z : F(\dots z \dots)$$
 Filler uses that variable
$$S/\text{NP}_z : \beta(\alpha) \rightarrow \text{NP} : \alpha \quad \text{VP}/\text{NP}_z : \beta$$
$$\text{VP}/\text{NP}_z : \beta(\alpha) \rightarrow \text{V} : \beta \quad \text{NP} : \alpha$$

*who*,  $\text{NP}[wh] : \lambda U.\lambda x.U(x) \wedge \mathbf{human}(x)$

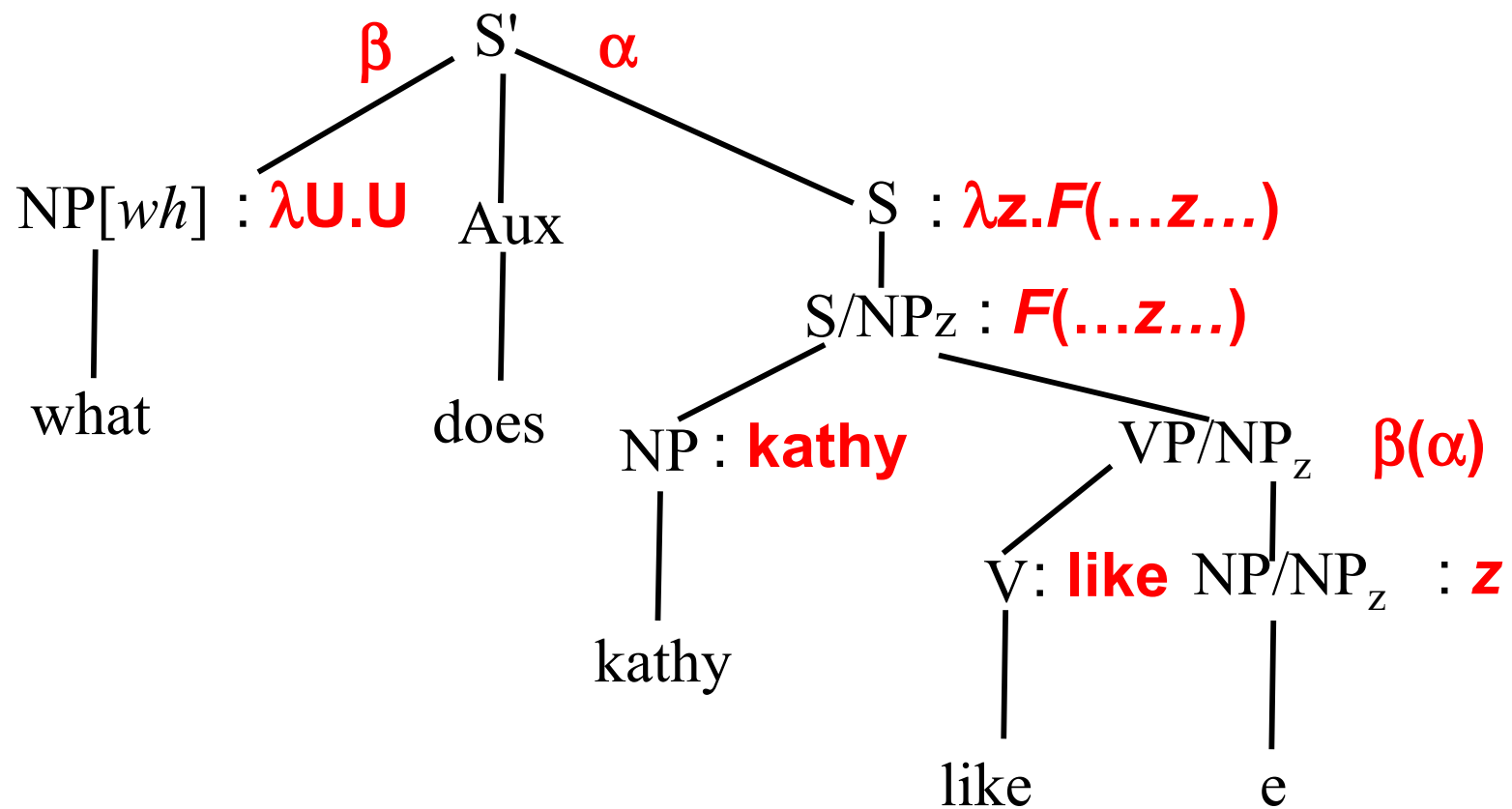
*what*,  $\text{NP}[wh] : \lambda U.U$

*which*,  $\text{Det}[wh] : \lambda P.\lambda V.\lambda x.P(x) \wedge V(x)$

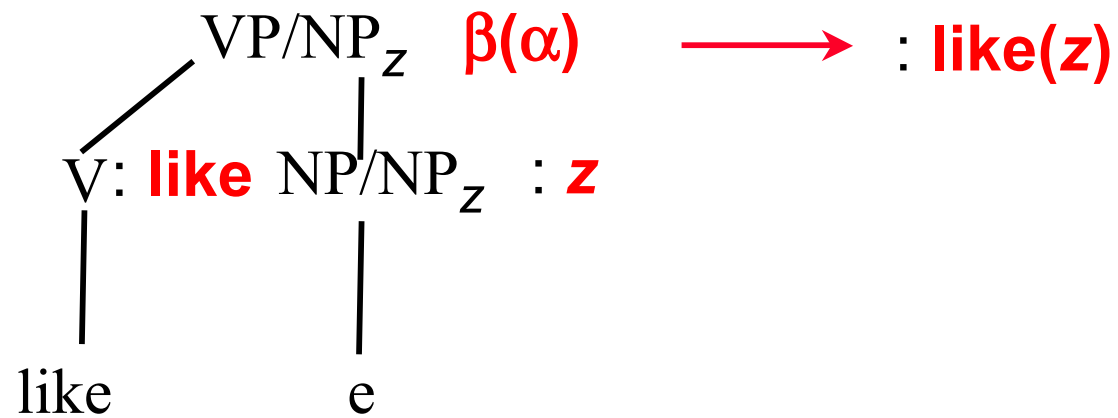
*how\_many*,  $\text{Det}[wh] : \lambda P.\lambda V.|\lambda x.P(x) \wedge V(x)|$

$|\cdot|$  is the operation that returns the cardinality of a set (count).

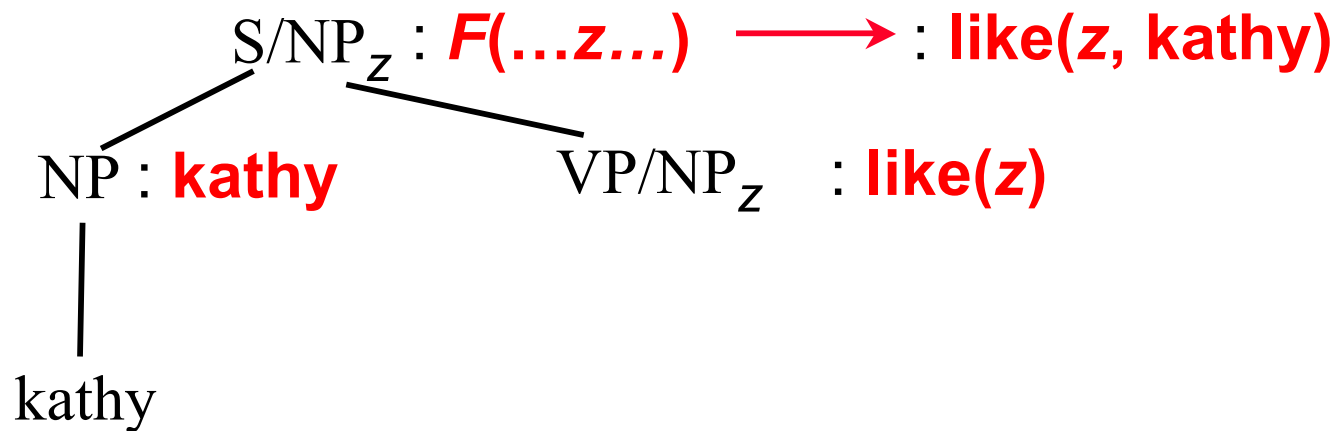
# Basic *wh* question: *what does kathy like*



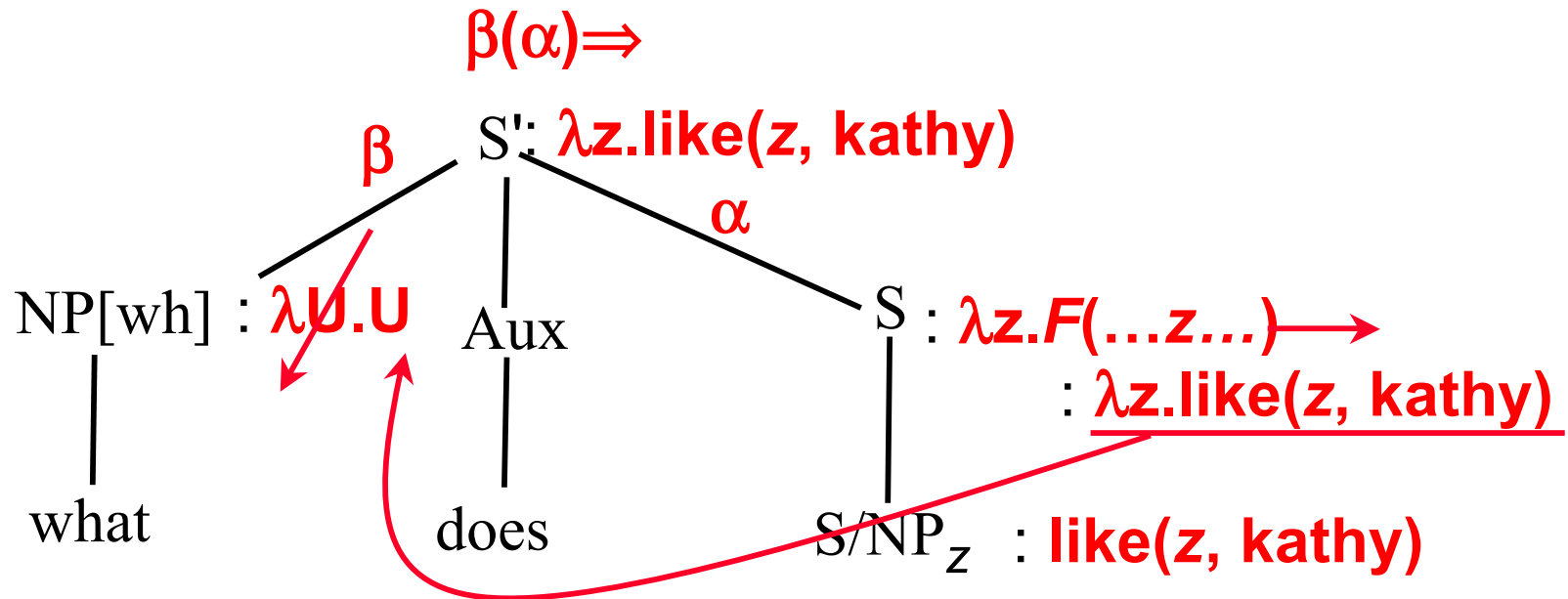
# Passing up the gap



Passing up gap variable to  $S/NP_z$ , then  
“S”

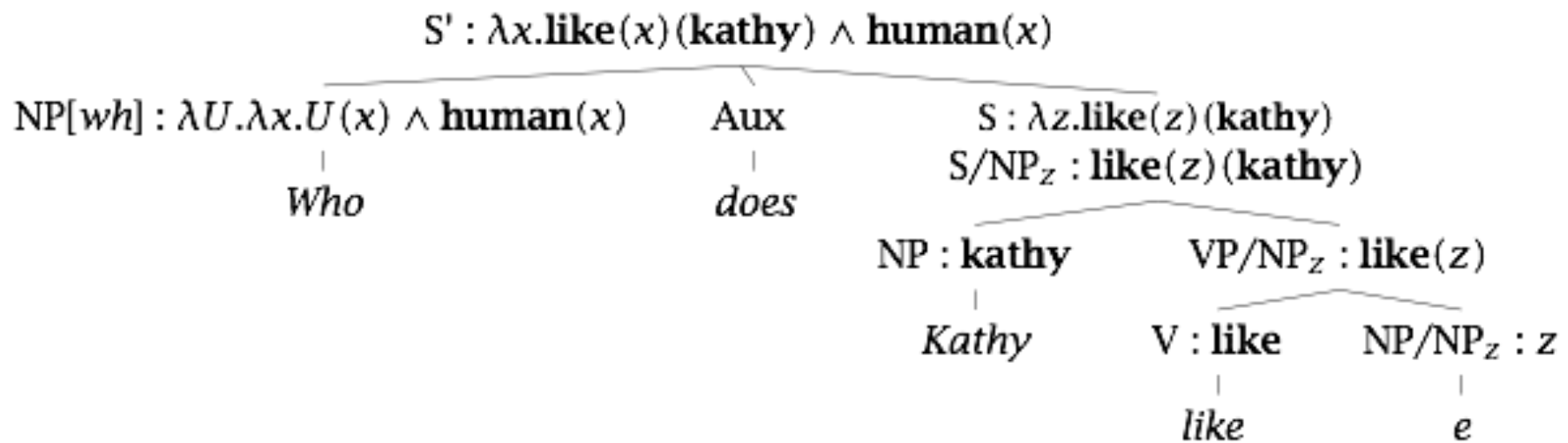


# Introducing gap variable & then link to filler *what*



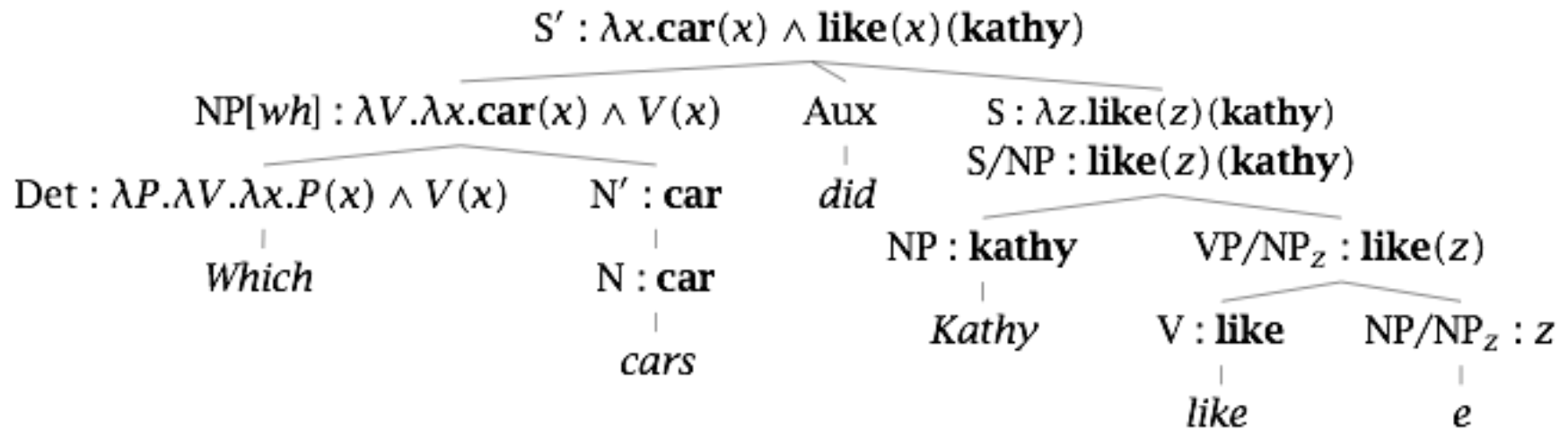
...Now an open proposition to check against the database:

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



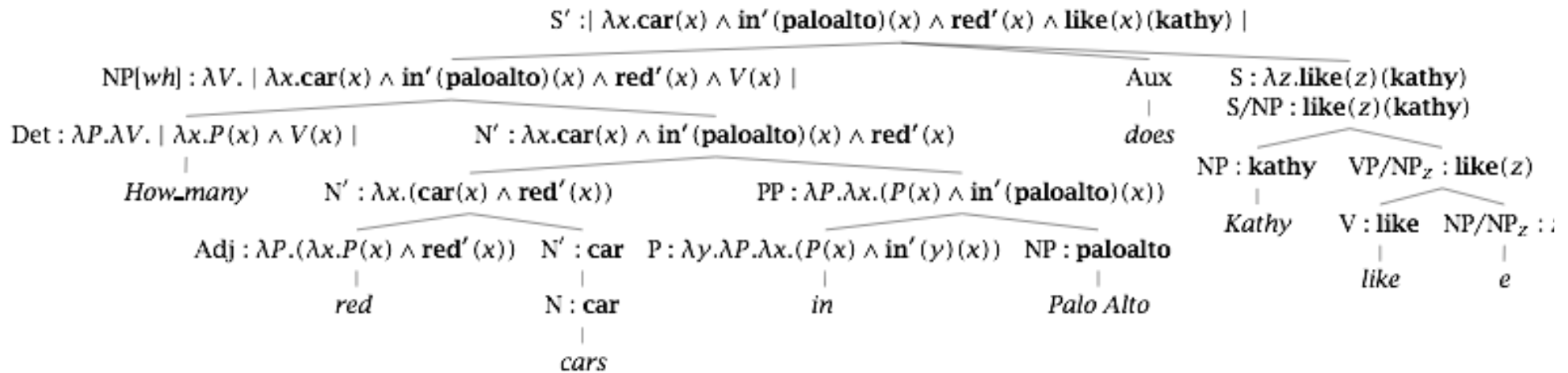
select liked from Likes, Humans where Likes.liker='Kathy' AND Humans.obj=Likes.liked

*Which car does Kathy like*



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

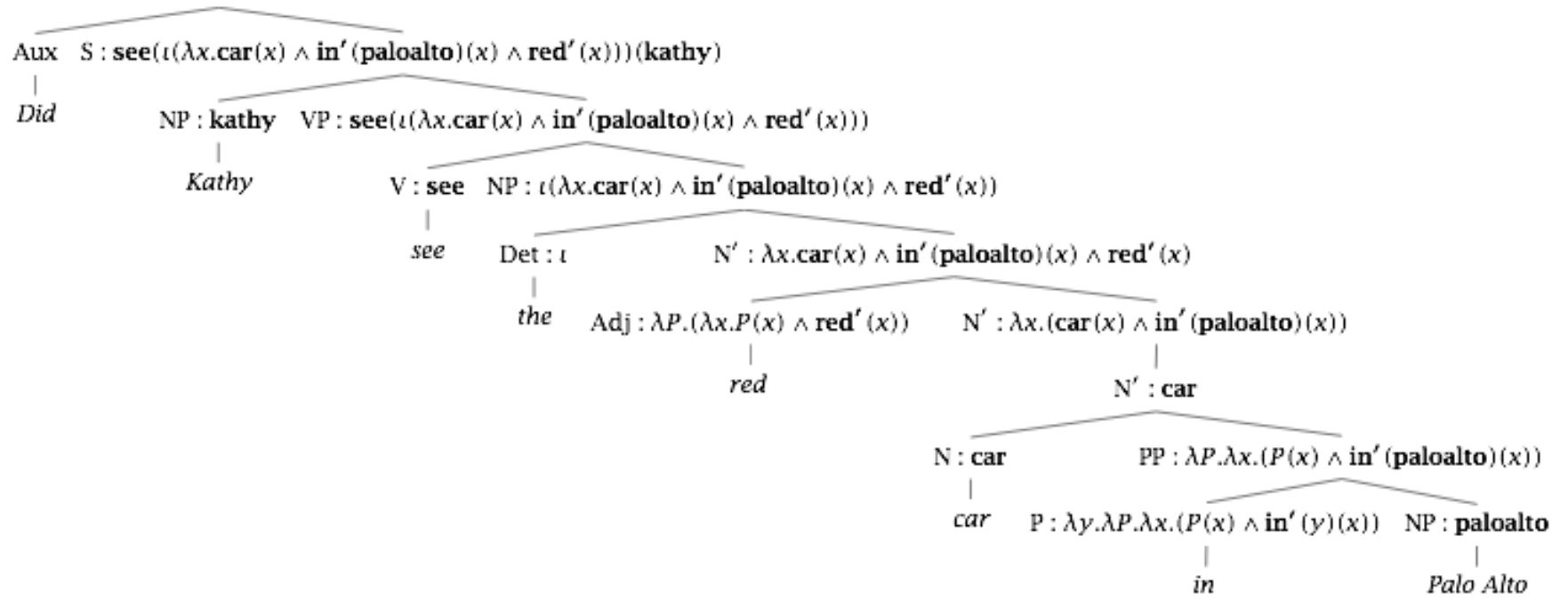
# How many cars in PaloAlto does Kathy like



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

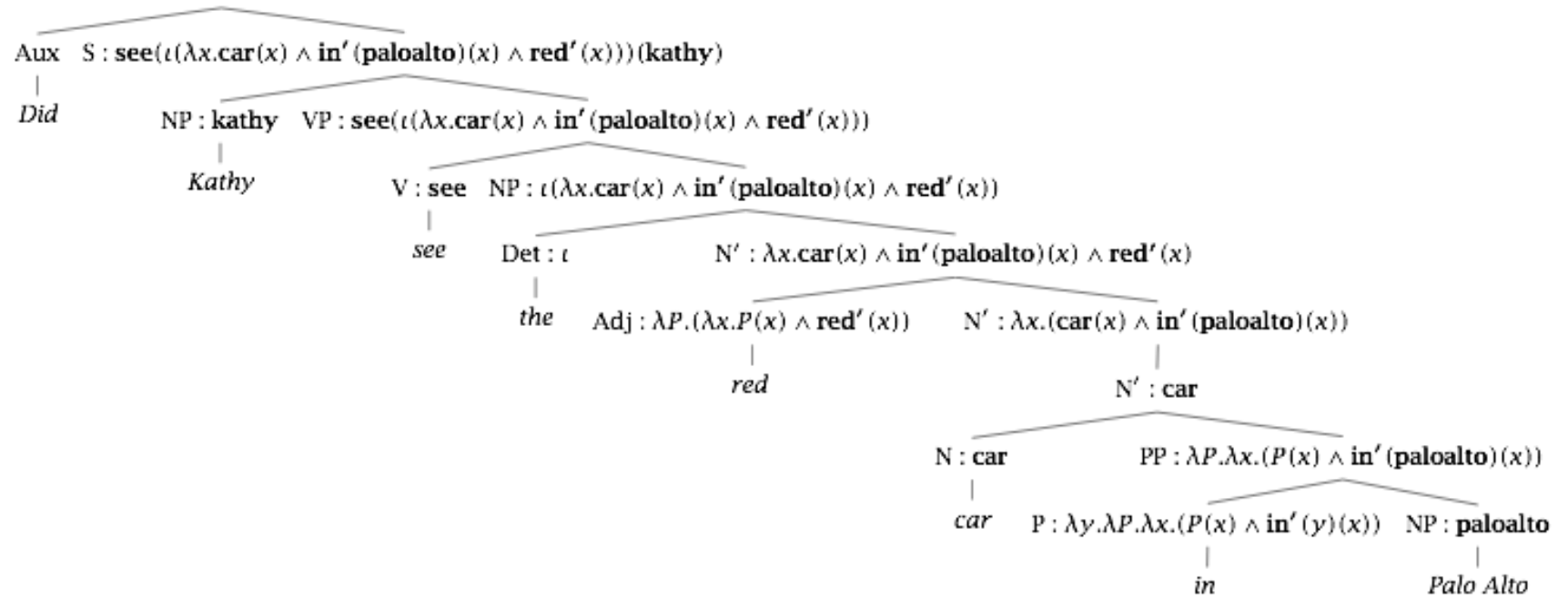
# *Did Kathy see the red car in PaloAlto* (ambig)

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



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)

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



# 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...
- Rocky likes no ice-cream...
- Plurals, tense, adverbs, adjectives, events,...

## What about determiners?

- Rocky chases a spy
- This doesn't mean *chases(rocky, spy)*
- Because this would just return only T or F – the same as:
  - *prime(17)*
  - *equal(4,2+2)*
  - *love(Obama,Palin)*
- What about “Rocky chases a spy and Bullwinkle chase a spy”

## The trouble with determiners

- What would the predicate calculus be for:
- *Every person* chases a spy; *A person* chases a spy

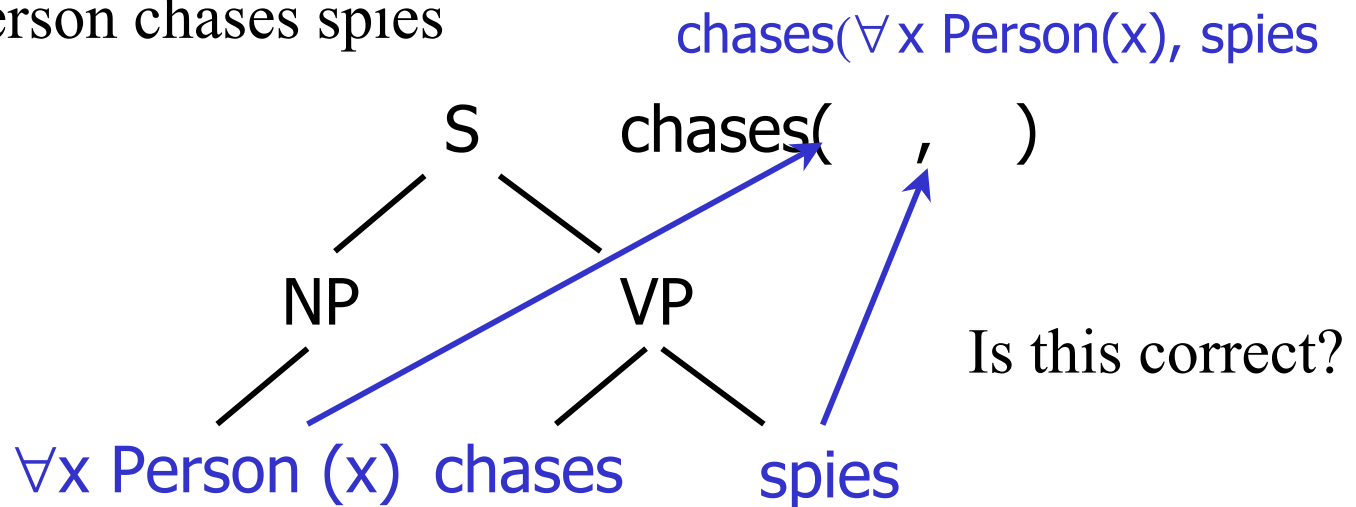
$\forall x (\text{Person } x \rightarrow \text{chases}(x, \text{spy}))$

$\exists x (\text{Person } x \ \& \ \text{chases}(x, \text{spy}))$

Let's try our  $\lambda$  trick out on this and see what happens...

# Quantifiers cause (apparent) problems

- If we apply composition directly following the syntax, what do we get?
- Every person chases spies



No! We want:

$\forall x$  (person(x)  $\rightarrow$  chases(x,spies))

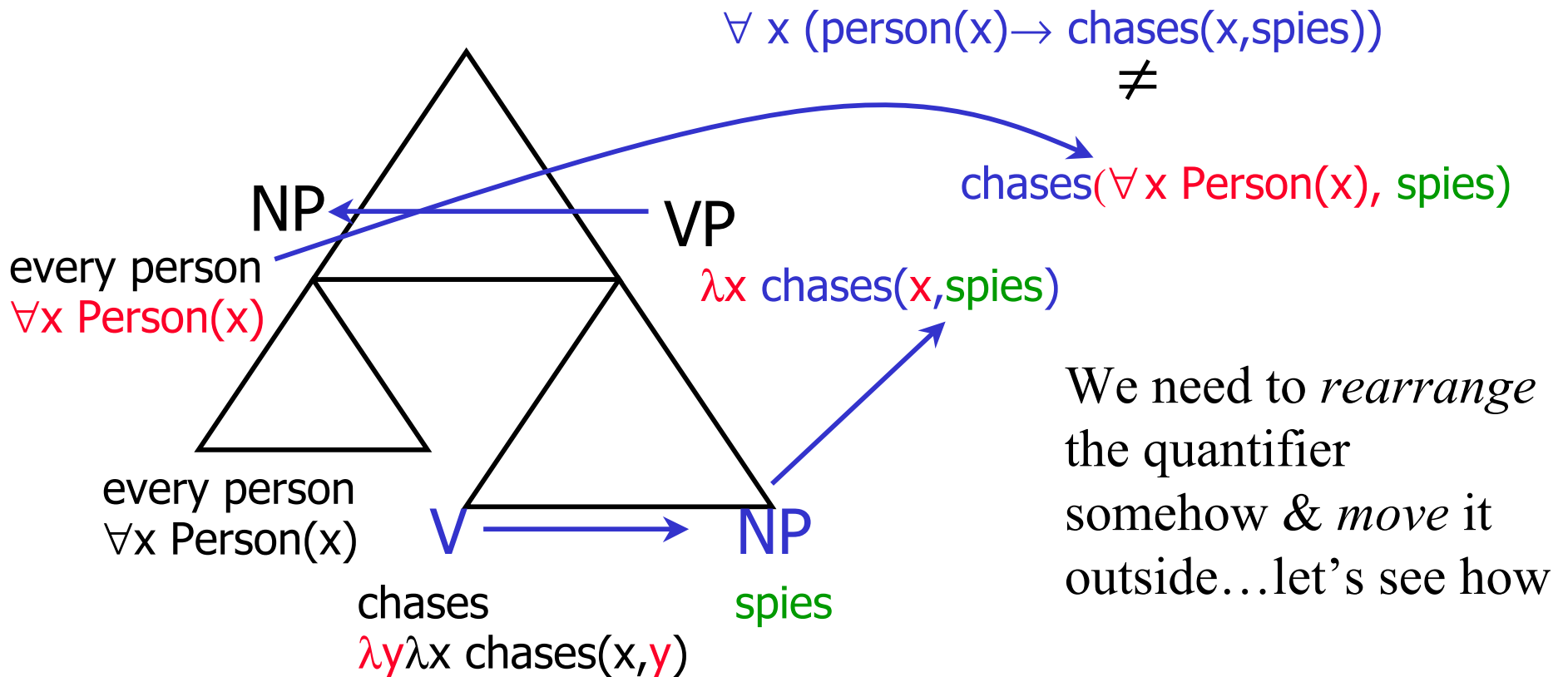
**BUT WE GET:**

Chases ( $\forall x$  Person(x), spies)

We need to rearrange the scope of the quantifier somehow!!!

Let's see how

# With quantifiers, we *lose* the direct substitution/isomorphism trick



Yields the wrong results!

“every person” is not something that seems to behave logically like the proper name in terms of its logical properties “rocky” - or does it?

In fact, we shall make proper names and quantified NPs look alike as far as their semantics goes...so we can treat them computationally in a uniform way

Let's see how

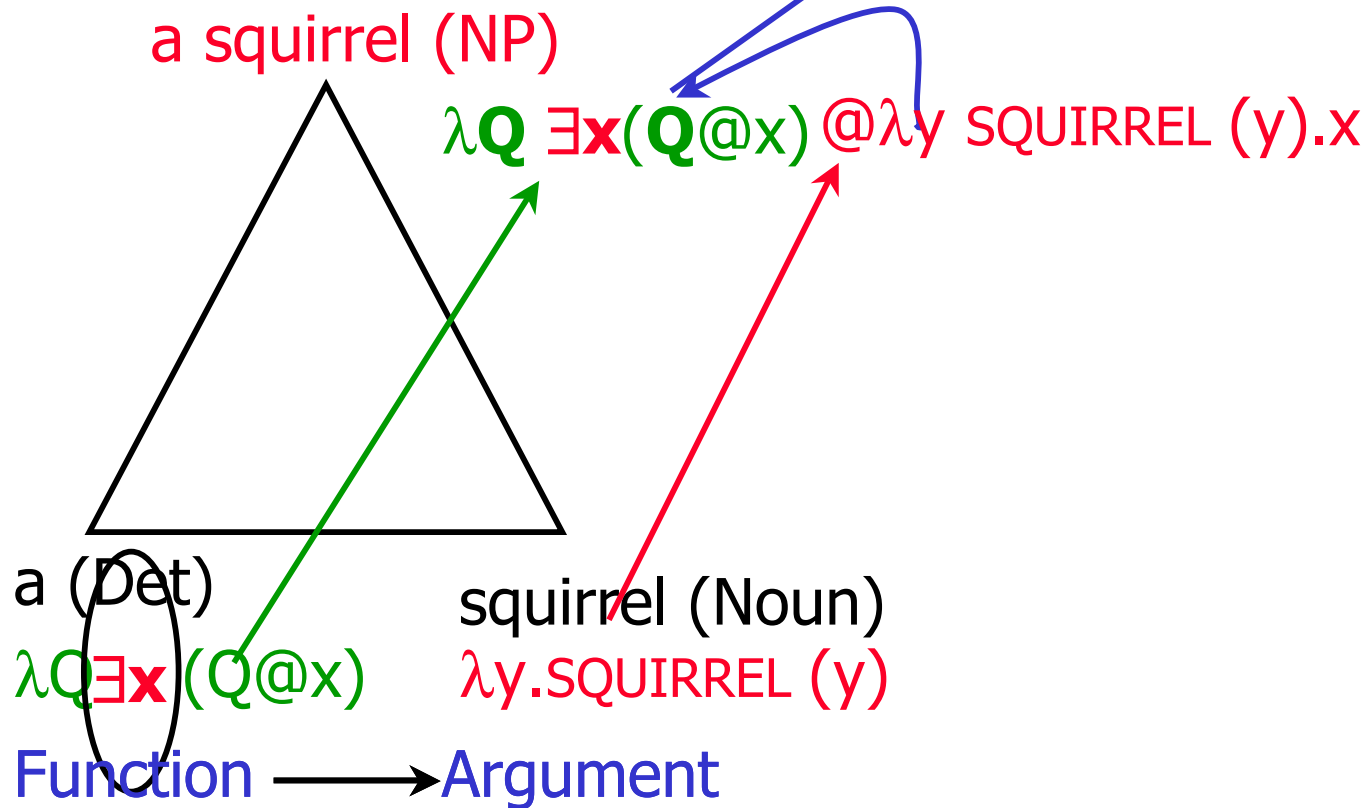
# Determiners

- “A squirrel sleeps”
- This has a simple logical form, if we follow our previous idea about ‘rocky sleeps’ - it is like ‘red squirrel’
- So we know what the result should look like:  
$$\exists x (\text{squirrel}(x) \ \& \ \text{sleeps}(x))$$
- Q: So how should we get this result?
- A: use same idea as before, abstract over property  $P$ :

## Now we have this for ‘a squirrel’...

- $\lambda P \exists x (\text{squirrel}(x) \ \& \ P(x))$
- We can replace  $P$  with any function of 1 argument, and we’ll get the right output form
- Same for universal quantifier ‘all’,  $\forall$
- $\lambda P \forall x (\text{squirrel}(x) \ \& \ P(x)) \equiv$  ‘all squirrels pee’
- But recall we can carry out one more level of abstraction... and abstract over the ‘property’ *squirrel*, call it ‘ $Q$ ’:  
 $\lambda Q \lambda P \exists x (Q(x) \ \& \ P(x))$
- This is the completely general form for *all* ‘quantified NPs’
- Note how it works: if we supply the predicate ‘squirrel’ first, it replaces the  $Q$ .
- Next, ‘sleeps’ replaces  $P$ .

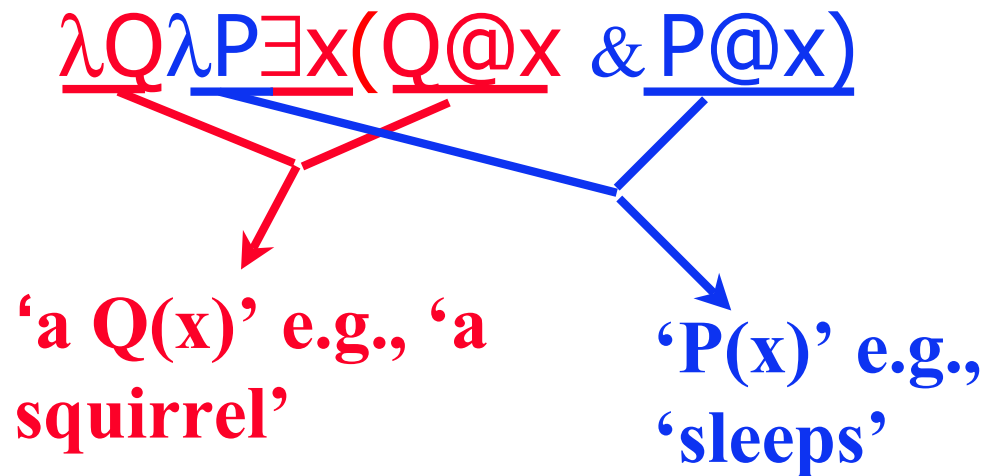
Now add  $\exists x$  to  
 template...to get  $\exists x$  Squirrel (x)  $\nearrow$   $\exists x$  SQUIRREL(x)  
 $\exists x \lambda y$  SQUIRREL (y)@x



## Next incorporate intransitive VP ‘sleeps’

- As before, this is just  $\lambda z \text{ sleep}(z)$
- Combines with NP, except now the Det-NP is the function, and the VP is the argument
- Remember the output goal template is:  
 $\exists x (\text{squirrel}(x) \wedge \text{sleeps}(x))$  and in general, the Determiner form looks like this:  
 $\exists x (Q(x) \wedge P(x))$
- And so far we have filled in:  
 $\exists x (\text{squirrel}(x) \dots)$
- So,  $P(x)$  must be filled in by the other predicate, from the VP,  
 $P(x) = \lambda z \text{ sleep}(z)$

So the full lambda form for the determiner ‘a’ looks like this, incorporating two ‘template’ lambdas, one for the determiner part of the NP, one for the VP

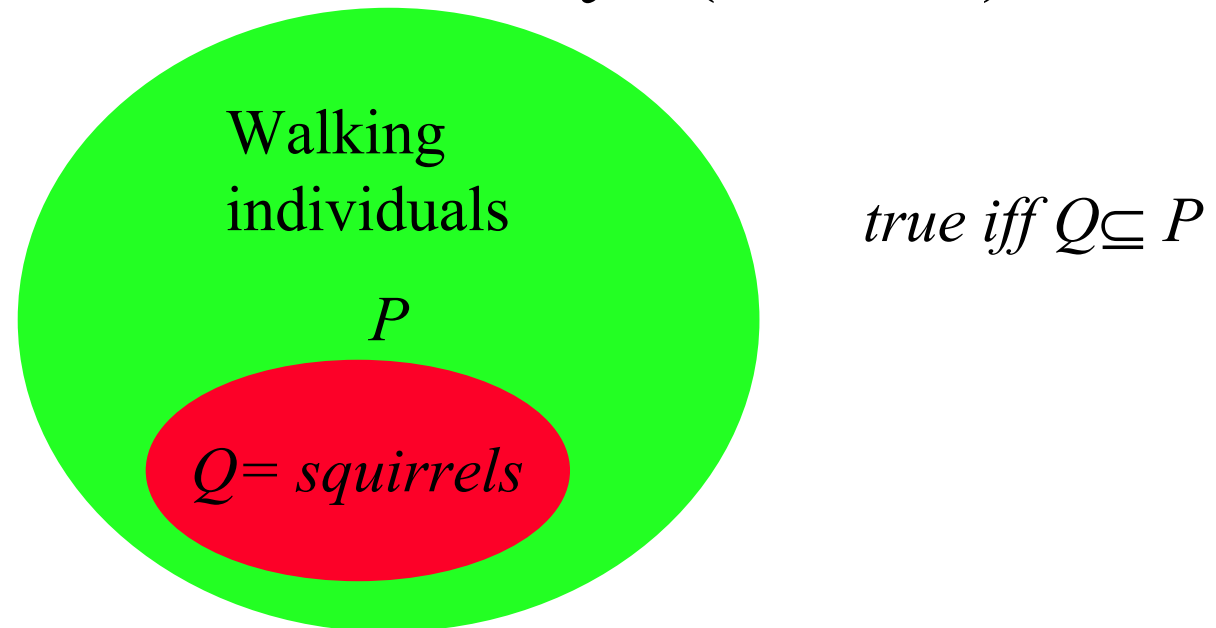


Note that a determiner is now a function that applies to the NP argument; resulting in another function that is applied to the verb... (rather than being the other way around). This is compatible with notion that a ‘Det’ is the head (functor) of a phrase. Just like a name.

## And in general...

- A Determiner  $D$  is thus a function of two arguments,  $P$  and  $Q$ , or, a two-place relation between sets (A function from a set to a set of sets)
- We will write this sometimes as  $D(A)(B)$ , where  $A, B$  are the two sets corresponding to the  $Q, P$ ; *note*  $Q, P$  are both of type  $\langle e, f \rangle$
- The  $Q$  part is the ‘squirrel’ portion and it itself a property, i.e. some subset of individuals in the domain (viz., the dogs); the  $P$  part is ‘barks’, another property, the subset of all barking individuals
- The output of this function is  $\{T, F\}$  (ie, of type  $t$ )
- We can write this simply as  $\text{EVERY/ALL}(Q)(P) = \text{true}$  iff  $Q \subseteq P$  (ie, the set of all dogs is contained in the set of all barking individuals)
- This will be useful as a ‘template’ for all sorts of determiners, what are sometimes called ‘generalized quantifiers’

# Picture for ‘every’ (or ‘all’)

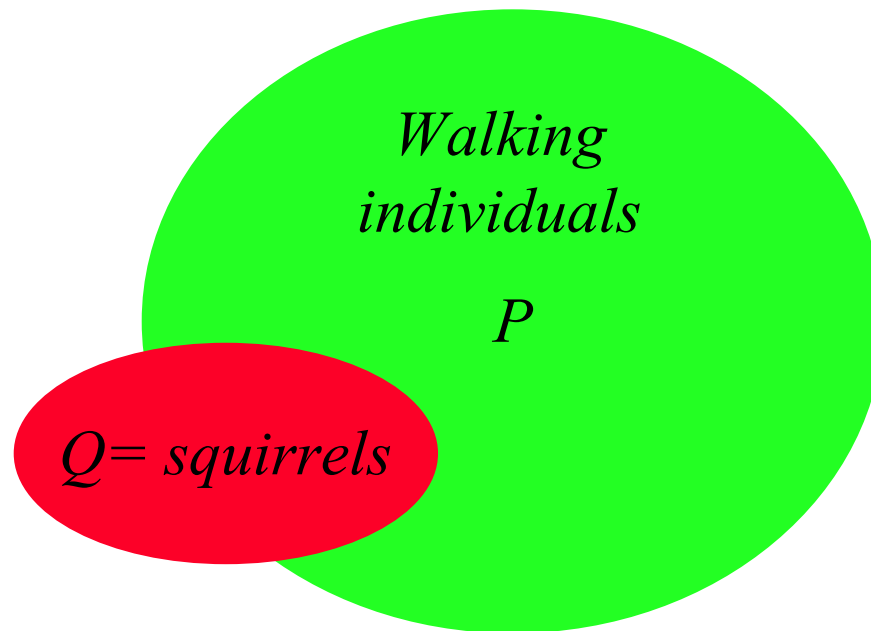


*Implemented as an SQL test, on tables representing  
 $Q$  and  $P$*

*Generalize: look at all the red, green relations...*

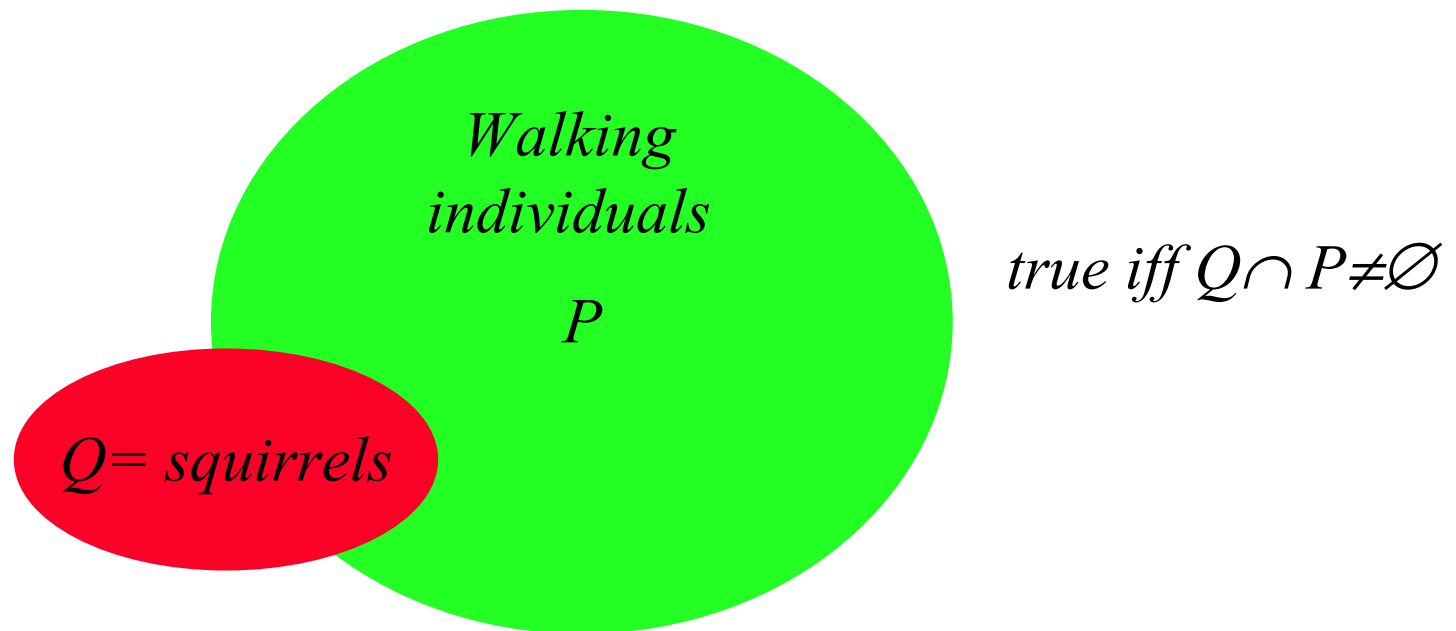
# “The ten squirrels walk”

*true iff  $|Q|=10$  &  $Q \cap P \neq \emptyset$*



We can write all determiners this way...  
*most, only five, etc.*

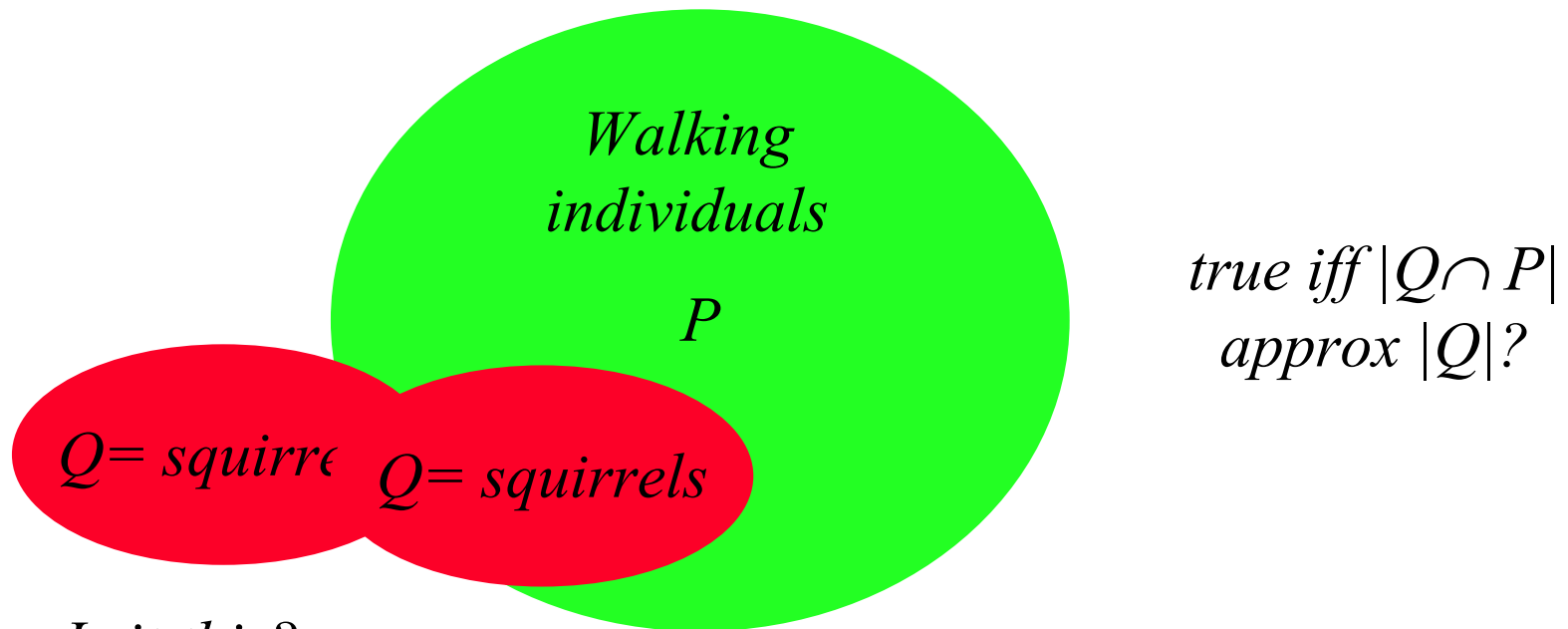
What about ‘some squirrels walk’ ?  
*What is the set relation now between  $Q$ ,  $P$ ?*



*(Of course,  $Q$  could be entirely contained in  $P$ )*  
*Again, the SQL translation is straightforward...*

# How about ‘most squirrels walk’?

*What is the set relation now between  $Q$ ,  $P$ ?*



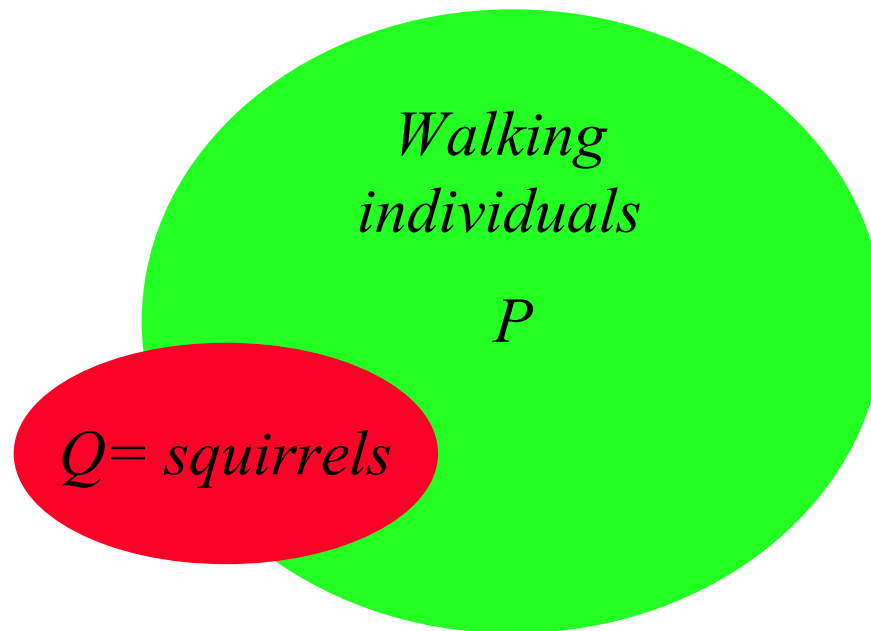
*true iff  $|Q \cap P|$   
approx  $|Q|$ ?*

*Is it this?*

*How about this?*

*Let's play this game a few more times...*

‘All but six squirrels walk’



*true iff*  $|Q| - |Q \cap P|$   
 $= 6$

*How long can we keep this game going?*

## A very long time: Lots of quantifiers!

- *At most two out of three, neither, some, a/an, no, several, more than six, at least six, exactly six, fewer than six, at most six, between six and ten, just finitely many, infinitely many, about a hundred, a couple of dozen, practically no, nearly twenty, approximately twenty, not more than ten, at least two and not more than ten, either fewer than five or else more than twenty, that many*

# Why should we care?

## Because these patterns interact with language

- A generalized quantifier  $GQ$  is said to be monotone increasing, also called upward entailing, just in case, for any two sets  $X$  and  $Y$  the following holds:  
if  $X \subseteq Y$  then  $GQ(X)$  entails  $GQ(Y)$
- The  $GQ$  *every squirrel* is monotone increasing. Why?
- the set of things that run fast is a subset of the set of things that run
- the first sentence below entails the second:
  1. Every squirrel flies fast
  2. Every squirrel flies

## Why should we care?

- A  $GQ$  is said to be monotone decreasing, also called downward entailing just in case, for any two sets  $X$  and  $Y$ , the following holds:  
if  $X \subseteq Y$  then  $GQ(Y)$  entails  $GQ(X)$
- Example: *no*
  1. No squirrel flies
  2. No squirrel flies fast

Why do we care?

*No* can license a ‘negative polarity item’ such as *any*

## Other such quantifiers: no, not, never

- John saw anything  
John didn't see anything
- I believe that she will budge an inch  
I don't believe that she will budge an inch
- Max said that he had ever been there before  
Max didn't say that he had ever been there before  
Max said that he hadn't ever been there before

## Another type of Determiner

A determiner (generalized quantifier)  $D$  is said to be conservative if the following equivalence holds for all  $A, B$ :

$$D(A)(B) \text{ iff } D(A)(A \cap B)$$

Every:

1. Every boy sleeps  $\equiv$
2. Every boy is a boy who sleeps

No:

1. No solution is perfect  $\equiv$
2. No solution is a perfect solution

Three:

1. Three circles are blue  $\equiv$
2. Three circles are blue circles

*Note:* both  $A$  and  $A \cap B$  are subsets of  $A$

To ‘compute’ *the* output, we only need to look at  $A$  and  $B$ , so the analysis is in a sense ‘local’

# Examples of non-conservative dets & how they differ

1. All but boys received a prize (but not ‘nall boys received a prize’)
2. Everyone but mothers attended

Compare:

3. Every mother attended

Q: in order to compute this last ‘set’, we look at all the mothers. Ditto if *all* boys received a prize. The denotation (set associated w/) of the head noun matters

But: what about sentences 1 or 2 ?

There we explicitly do not look at the set of ‘boys’ or ‘mothers’ – but rather outside these sets

That is the sense in which these Determiners are nonconservative: they don’t ‘stay inside’ the head noun; the head noun does not determine the range of quantification

Non-conservative Determiners violate the constraint:  $A$  and  $A \cap B$  are subsets of  $A$

They are ‘context sensitive’ (or rather nonlocal)

# Are there any non-conservative determiners?

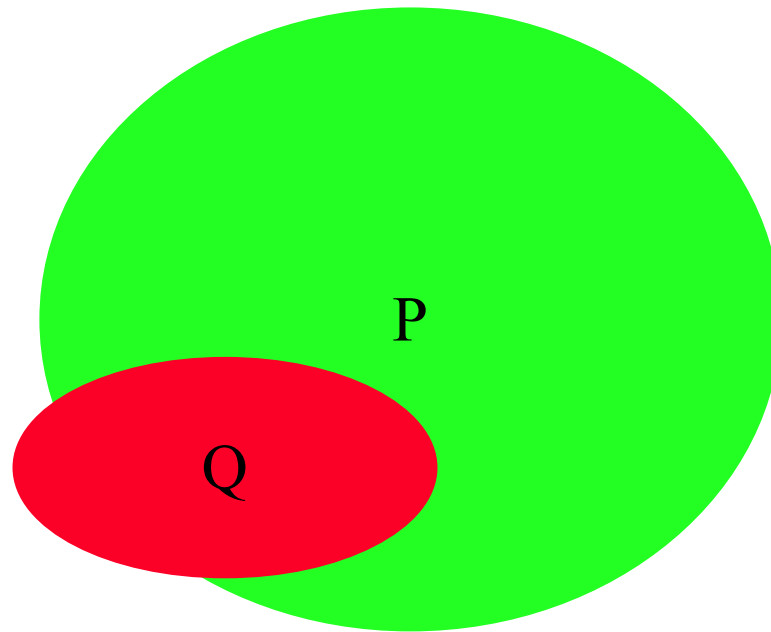
*Only*:

1. Only males are astronauts (false) ≠ Only males are male astronauts (true)

Arguable that *only* is a determiner; it is a ‘focus sensitive’ adverb

# What about ‘all but’???

- Can we do it just by playing around with:



Answer: NO!!

# Note definition of *everyone except, all but etc*

- *All but* (X)(Y) iff  $(A-X) \subseteq Y$

Depends on another set  
A, something other  
than  
X and Y

Note that everything but, all but, etc. are not even syntactic constituents, let alone natural language determiners!

# A semantic constraint

- There seems to be no quantifier, call it ‘nall’ that is defined formally as ‘all but’ in any natural language
- Why???

# In fact, the natural language Determiners are quite constrained!

- Conjecture: Natural language determiners are conservative
- **Theorem** (Keenan & Stavi, 1986). Starting from *every* and *a* as basic determiners, and building other determiners by Boolean operations of negation, conjunction, and disjunction, the resulting set of determiners consists of exactly the conservative determiners
- Conjecture: is this Boolean apparatus the basic tool behind concept formation?