L100: Lecture 7, Compositional semantics

Ann Copestake

Computer Laboratory
University of Cambridge

November 2012

Outline of today's lecture

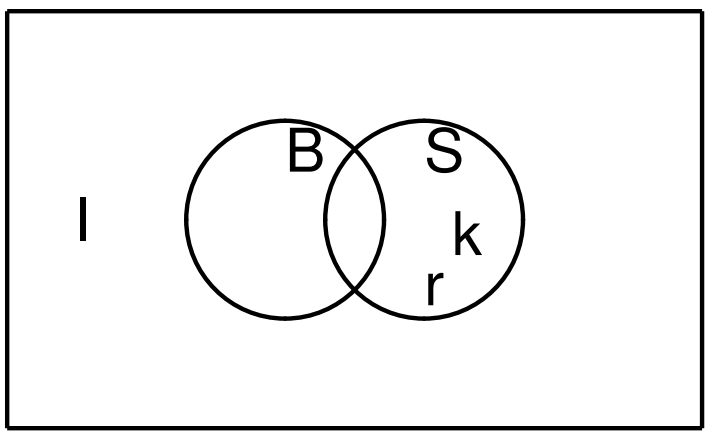
- Model-theoretic semantics and denotation
- Quantifiers
- The semantics of some nominal modifiers
- General principles of semantic composition
- Typing in compositional semantics
- Lambda expressions
- Example grammar with lambda calculus
- Quantifiers again

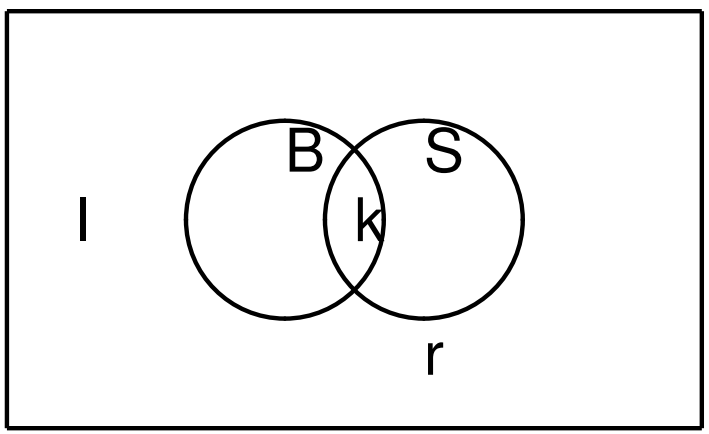
Model-theoretic semantics

Kitty sleeps is true in a particular model if the individual denoted by Kitty (say *k*) in that model is a member of the set denoted by *sleep* (*S*):

$$k \in S$$

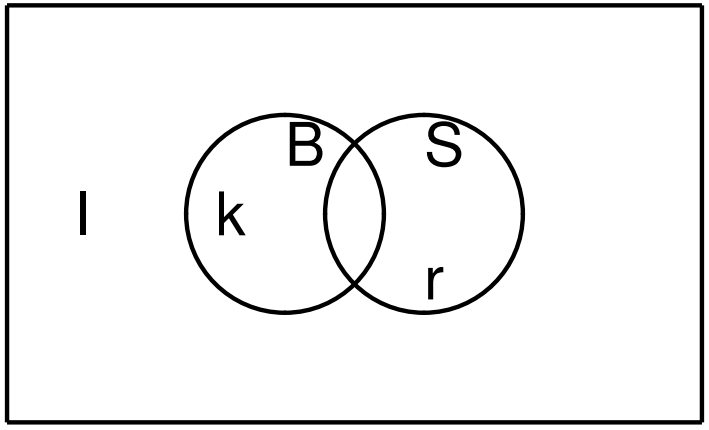
Two models where *Kitty sleeps* is true:



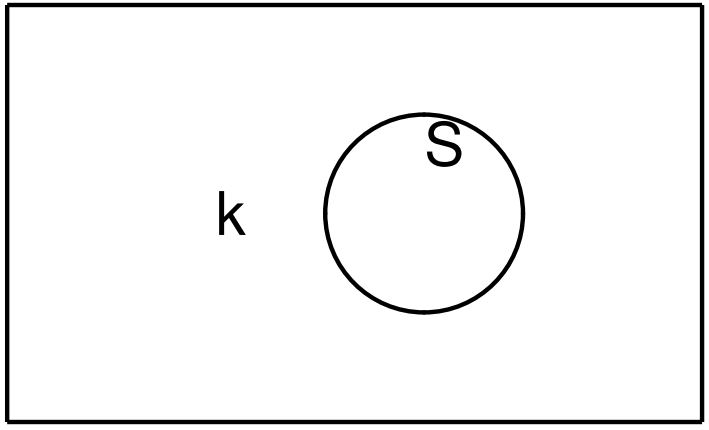


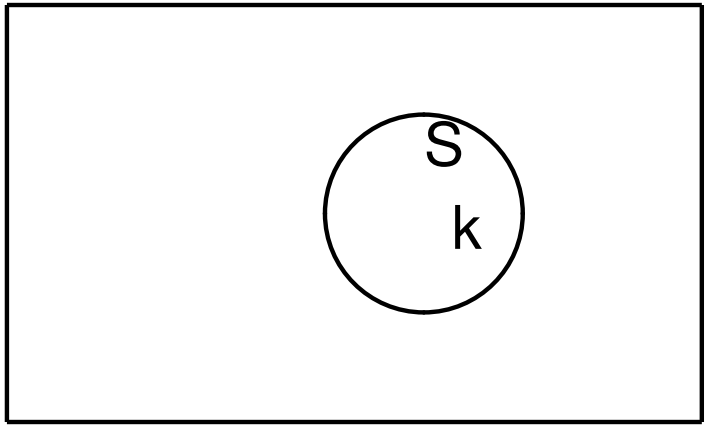
Model-theoretic semantics

A model where *Kitty sleeps* is false:



Only showing relevant entities:





Ordered pairs

- The denotation of *chase* is a set of ordered pairs.
- For instance, if Kitty chases Rover and Lynx chases Rover and no other chasing occurs then *chase* denotes $\{\langle k, r\rangle, \langle l, r\rangle\}$.
- Ordered pairs are not the same as sets.
  - Repeated elements: if *chase* denotes $\{\langle r, r\rangle\}$ then Rover chased himself.
  - Order is significant, $\langle k, r\rangle$ is not the same as $\langle r, k\rangle$ 'Kitty chased Rover' vs 'Rover chased Kitty'

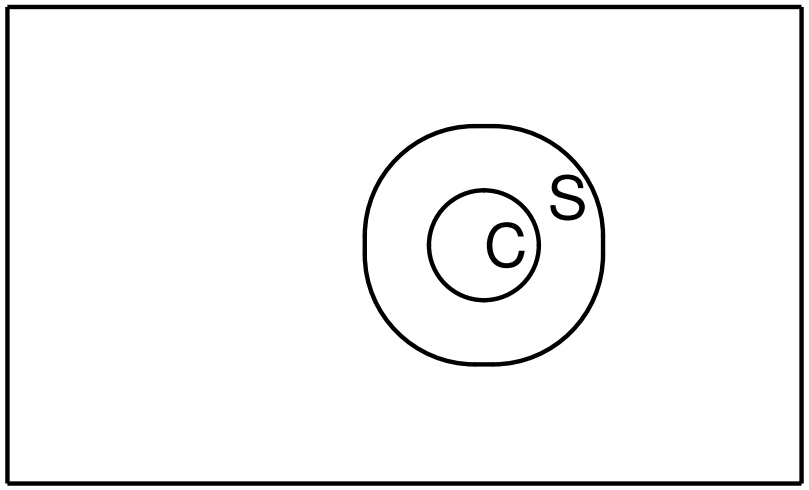
every, some and no

The sentence *every cat sleeps* is true just in case the set of all cats is a subset of the set of all things that sleep.
If the set of cats is $\{k, l\}$ then *every cat sleeps* is equivalent to:

$$\{k, l\} \subseteq S$$

Or, if we name the set of cats C:

$$C \subseteq S$$



Logic and model theory

- Model theory: meaning can be expressed set theoretically with respect to a model.
- But set theoretic representation is messy: we want to abstract away from individual models.
- Instead, think in terms of logic: truth-conditions for *and*, *or* etc. e.g., if *Kitty sleeps* is true, then *Kitty sleeps or Rover barks* will necessarily also be true.

Quantifiers

(1) $\exists x[\text{sleep}'(x)]$
Something sleeps

(2) $\forall x[\text{sleep}'(x)]$
Everything sleeps

(3) $\exists x[\text{cat}'(x) \wedge \text{sleep}'(x)]$
Some cat sleeps

(4) $\forall x[\text{cat}'(x) \implies \text{sleep}'(x)]$
Every cat sleeps

(5) $\forall x[\text{cat}'(x) \implies \exists y[\text{chase}'(x, y)]]$
Every cat chases something

(6) $\forall x[\text{cat}'(x) \implies \exists y[\text{dog}'(y) \wedge \text{chase}'(x, y)]]$
Every cat chases some dog

Variables

- x, y, z pick out entities in model according to variable assignment function: e.g., sleeps$'(x)$ may be true or false in a particular model, depending on the function.
- Constants and variables:

  (29) $\forall x[\text{cat}'(x) \implies \text{chase}'(x, r)]$
  Every cat chases Rover.
- No explicit representation of variable assignment function: we just care about bound variables for now (i.e., variables in the scope of a quantifier).

Quantifier scope ambiguity

- The truth conditions of formulae with quantifiers depend on the relative scope of the quantifiers
- Natural languages sentences can be ambiguous wrt FOPC without being syntactically ambiguous
- *Everybody in the room speaks two languages*

same two languages or not?

The semantics of some nominal modifiers

(33) every big cat sleeps

$$\forall x[(\text{cat}'(x) \wedge \text{big}'(x)) \implies \text{sleep}'(x)]$$

(34) every cat on some mat sleeps
wide scope *every*:

$$\forall x[(\text{cat}'(x) \wedge \exists y[\text{mat}'(y) \wedge \text{on}'(x,y)]) \implies \text{sleep}'(x)]$$

wide scope *some* (i.e., single mat):

$$\exists y[\text{mat}'(y) \wedge \forall x[(\text{cat}'(x) \wedge \text{on}'(x,y)) \implies \text{sleep}'(x)]]$$

$\text{on}'(x,y)$ must be in the scope of both quantifiers.

Adjectives and prepositional phrases (in this use) are syntactically modifiers.
Semantically: intersective modifiers: combine using $\wedge$, modified phrase denotes a subset of what's denoted by the noun.

Going from FOPC to natural language

Well-formed FOPC expressions, don't always correspond to natural NL utterances. For instance:

(35) $\forall x[\text{cat}'(x) \wedge \exists y[\text{bark}'(y)]]$

This best paraphrase of this I can come up with is:

(36) Everything is a cat and there is something which barks.

Semantic composition

- Semantic rules parallel syntax rules.
- Semantics is build up compositionally: meaning of the whole is determined from the meaning of the parts.
- Semantic derivation: constructing the semantics for a sentence.
- Interpretation with respect to a model (true or false).
- The logical expressions constructed (logical form) could (in principle) be dispensed with.
  Maybe . . .

Semantic composition

- Semantic rules parallel syntax rules.
- Semantics is build up compositionally: meaning of the whole is determined from the meaning of the parts.
- Semantic derivation: constructing the semantics for a sentence.
- Interpretation with respect to a model (true or false).
- The logical expressions constructed (logical form) could (in principle) be dispensed with.
  Maybe . . .

Typing

- Semantic typing ensures that semantic expressions are consistent.
  e.g., chase′(dog′(k)) is ill-formed.
- Two basic types:
  - e is the type for entities in the model (such as k)
  - t is the type for truth values (i.e., either 'true' or 'false')

  All other types are composites of the basic types.
- Complex types are written $\langle$type1, type2$\rangle$, where type1 is the argument type and type2 is the result type and either type1 or type2 can be basic or complex.
  e.g., $\langle e, \langle e, t\rangle\rangle$, $\langle t, \langle t, t\rangle\rangle$

Typing

- Semantic typing ensures that semantic expressions are consistent.
  e.g., $\text{chase}'(\text{dog}'(k))$ is ill-formed.
- Two basic types:
  - e is the type for entities in the model (such as k)
  - t is the type for truth values (i.e., either 'true' or 'false')

  All other types are composites of the basic types.
- Complex types are written $\langle \text{type1}, \text{type2} \rangle$, where type1 is the argument type and type2 is the result type and either type1 or type2 can be basic or complex.
  e.g., $\langle e, \langle e, t \rangle \rangle$, $\langle t, \langle t, t \rangle \rangle$

Typing

- Semantic typing ensures that semantic expressions are consistent.
  e.g., $\text{chase}'(\text{dog}'(k))$ is ill-formed.
- Two basic types:
  - e is the type for entities in the model (such as k)
  - t is the type for truth values (i.e., either 'true' or 'false')

  All other types are composites of the basic types.
- Complex types are written $\langle \text{type1}, \text{type2} \rangle$, where type1 is the argument type and type2 is the result type and either type1 or type2 can be basic or complex.
  e.g., $\langle e, \langle e, t \rangle \rangle$, $\langle t, \langle t, t \rangle \rangle$

Types of lexical entities

First approximation: predicates corresponding to:

- intransitive verbs (e.g. bark′) — $\langle e, t \rangle$
  take an entity and return a truth value
- (simple) nouns (e.g., dog′, cat′) — $\langle e, t \rangle$
- transitive verbs (e.g., chase′) — $\langle e, \langle e, t \rangle \rangle$
  take an entity and return something of the same type as an intransitive verb

Lambda expressions

Lambda calculus is a logical notation to express the way that predicates 'look' for arguments. e.g.,

$$\lambda x[\text{bark}'(x)]$$

- Syntactically, λ is like a quantifier in FOPC: the lambda variable (x above) is said to be within the scope of the lambda operator
- lambda expressions correspond to functions: they denote sets (e.g., $\{x : x \text{ barks}\}$)
- the lambda variable indicates a variable that will be bound by function application.

Lambda conversion

Applying a lambda expression to a term will yield a new term, with the lambda variable replaced by the term (lambda-conversion).
For instance:

$$\lambda x[\text{bark}'(x)](k) = \text{bark}'(k)$$

Lambda conversion and typing

Lambda conversion must respect typing, for example:

$$\begin{array}{lll} \lambda x[\text{bark}'(x)] & k & \text{bark}'(k) \\ \langle e, t \rangle & e & t \end{array}$$

$$\lambda x[\text{bark}'(x)](k) = \text{bark}'(k)$$

We cannot combine expressions of incompatible types. e.g.,

$$\lambda x[\text{bark}'(x)](\lambda y[\text{snore}'(y)])$$

is not well-formed

Multiple variables

If the lambda variable is repeated, both instances are instantiated:

$$\underset{\langle e,t\rangle}{\lambda x[\text{bark}'(x) \wedge \text{sleep}'(x)]} \quad \underset{e}{\text{r}} \quad \underset{t}{\text{bark}'(r) \wedge \text{sleep}'(r)}$$

$\lambda x[\text{bark}'(x) \wedge \text{sleep}'(x)]$ denotes the set of things that bark and sleep

$$\lambda x[\text{bark}'(x) \wedge \text{sleep}'(x)](r) = \text{bark}'(r) \wedge \text{sleep}'(r)$$

Transitive and intransitive verbs

A partially instantiated transitive verb predicate is of the same type as an intransitive verb:

$$\begin{array}{lll} \lambda x[\text{chase}'(x, r)] & k & \text{chase}'(k, r) \\ \langle e, t \rangle & e & t \end{array}$$

$\lambda x[\text{chase}'(x, r)]$ is the set of things that chase Rover.

$$\lambda x[\text{chase}'(x, r)](k) = \text{chase}'(k, r)$$

Transitive verbs

Lambdas can be nested: transitive verbs can be represented so they apply to only one argument at once.
For instance:

$$\lambda x[\lambda y[\text{chase}'(y, x)]]$$

often written

$$\lambda x \lambda y[\text{chase}'(y, x)]$$

$$\lambda x[\lambda y[\text{chase}'(y, x)]](r) = \lambda y[\text{chase}'(y, r)]$$

Bracketing shows the order of application in the conventional way:

$$\begin{aligned}(\lambda x[\lambda y[\text{chase}'(y, x)]](r))(k) = \lambda y[\text{chase}'(y, r)](k) \\ = \text{chase}'(k, r)\end{aligned}$$

A simple grammar

`S -> NP VP`
$\mathrm{VP}'(\mathrm{NP}')$

`VP -> Vtrans NP`
$\mathrm{Vtrans}'(\mathrm{NP}')$

`VP -> Vintrans`
$\mathrm{Vintrans}'$

`Vtrans -> chases`
$\lambda x \lambda y[\mathrm{chase}'(y, x)]$

`Vintrans -> barks`
$\lambda z[\mathrm{bark}'(z)]$

`Vintrans -> sleeps`
$\lambda w[\mathrm{sleep}'(w)]$

`NP -> Kitty`
k

`NP -> Lynx`
l

`NP -> Rover`
r

Example 1: lambda calculus with transitive verbs

1. `Vtrans -> chases`
   $\lambda x \lambda y[\text{chase}'(y, x)]$ (type: $\langle e, \langle e, t \rangle \rangle$)
2. `NP -> Rover`
   r (type: e)
3. `VP -> Vtrans NP`
   $\text{Vtrans}'(\text{NP}')$
   $\lambda x \lambda y[\text{chase}'(y, x)](r)$
   $= \lambda y[\text{chase}'(y, r)]$ (type: $\langle e, t \rangle$)
4. `NP -> Lynx`
   l (type: e)
5. `S -> NP VP`
   $\text{VP}'(\text{NP}')$
   $\lambda y[\text{chase}'(y, r)](l)$
   $= \text{chase}'(l, r)$ (type: t)

Ditransitive verbs

The semantics of *give* can be represented as $\lambda x \lambda y \lambda z[\text{give}'(z, y, x)]$. The ditransitive rule is:

```
VP -> Vditrans NP1 NP2
```

$(Vditrans'(NP1'))(NP2')$

Two lambda applications in one rule:

$$\begin{aligned}(\lambda x[\lambda y[\lambda z[\text{give}'(z, y, x)]]](l))(r) \\ = \lambda y[\lambda z[\text{give}'(z, y, l)]](r) \\ = \lambda z[\text{give}'(z, r, l)]\end{aligned}$$

Here, indirect object is picked up first (arbitrary decision in rule/semantics for *give*)

Ditransitive verbs with PP

PP form of the ditransitive uses the same lexical entry for *give*, but combines the arguments in a different order:

```
VP -> Vditrans NP1 PP
```

$(Vditrans'(PP'))(NP1')$

Example 2

Rover gives Lynx Kitty

1. `Vditrans -> gives`
   $\lambda x[\lambda y[\lambda z[\text{give}'(z, y, x)]]]$ type: $\langle e, \langle e, \langle e, t\rangle\rangle\rangle$
2. `NP -> Lynx`
   l type: e
3. `NP -> Kitty`
   k type: e
4. `VP -> Vditrans NP1 NP2`
   $(\text{Vditrans}'(\text{NP1}'))(\text{NP2}')$
   $(\lambda x[\lambda y[\lambda z[\text{give}'(z, y, x)]]](l))(k)$
   $= \lambda y[\lambda z[\text{give}'(z, y, l)]](k)$ type: $\langle e, \langle e, t\rangle\rangle$
   $= \lambda z[\text{give}'(z, k, l)]$ type: $\langle e, t\rangle$

Example 2, continued

5 `NP -> Rover`
r type: e

6 `S -> NP VP`
$\text{VP}'(\text{NP}')$
$= \lambda z[\text{give}'(z, k, l)](r)$
$= \text{give}'(r, k, l)$ type: t

PP ditransitive: Exercise

Rover gives Kitty to Lynx

Assumptions:

- has the same semantics as
  Rover gives Lynx Kitty
- No semantics associated with *to*
- Same lexical entry for *give* as for the NP case
- So difference has to be in the VP rule

Coordination

`S[conj=yes] -> CONJ S1[conj=no]`
$CONJ'(S1')$

`S[conj=no] -> S1[conj=no] S2[conj=yes]`
$S2'(S1')$

`CONJ -> and`
$\lambda P[\lambda Q[P \wedge Q]]$ *type:* $\langle t, \langle t, t \rangle \rangle$

`CONJ -> or`
$\lambda P[\lambda Q[P \vee Q]]$ *type:* $\langle t, \langle t, t \rangle \rangle$

Example 3: lambda calculus and coordination

Lynx chases Rover or Kitty sleeps

1. `CONJ -> or`
   $\lambda P[\lambda Q[P \vee Q]]$
2. `S[conj=yes] -> CONJ S1[conj=no]`
   $\mathsf{CONJ}'(\mathsf{S1}')$
   $\lambda P[\lambda Q[P \vee Q]](\mathsf{sleep}'(k)) = \lambda Q[\mathsf{sleep}'(k) \vee Q]$
3. `S[conj=no] -> S1[conj=no] S2[conj=yes]`
   $\mathsf{S2}'(\mathsf{S1}')$
   $\lambda Q[\mathsf{sleep}'(k) \vee Q](\mathsf{chase}'(l, r)) = \mathsf{sleep}'(k) \vee \mathsf{chase}'(l, r)$

VP coordination

- sentential conjunctions are of the type $\langle t, \langle t, t \rangle \rangle$
- conjunctions can also combine VPs, so $\langle \langle e, t \rangle, \langle \langle e, t \rangle, \langle e, t \rangle \rangle \rangle$: conjunctions are of polymorphic type
- general schema for conjunctions is $\langle type, \langle type, type \rangle \rangle$.

VP conjunction rule uses the same lexical entries for *and* and *or* as sentential conjunction:

```
VP[conj=yes] -> CONJ VP1[conj=no]
```

$\lambda R[\lambda x[(\text{CONJ}'(R(x)))(\text{VP1}'(x))]]$

```
VP[conj=no] -> VP1[conj=no] VP2[conj=yes]
```

$\text{VP2}'(\text{VP1}')$

This looks complicated, but doesn't use any new formal devices.

Example 4

1. chases Rover
   $\lambda y[\text{chase}'(y, r)]$
2. `CONJ -> and`
   $\lambda P \lambda Q[P \wedge Q]$
3. and chases Rover
   `VP[conj=yes] -> CONJ VP1[conj=no]`
   $\lambda R \lambda x[(\text{CONJ}'(R(x)))(\text{VP1}'(x))]$ (grammar rule)
   $\lambda R \lambda x[(\lambda P[\lambda Q[P \wedge Q]](R(x)))(\lambda y[\text{chase}'(y, r)](x))]$
   (substituted CONJ and VP1)
   $= \lambda R \lambda x[(\lambda P[\lambda Q[P \wedge Q]](R(x)))(\text{chase}'(x, r))]$ (lambda y)
   $= \lambda R \lambda x[\lambda Q[R(x) \wedge Q](\text{chase}'(x, r))]$ (applied lambda P)
   $= \lambda R \lambda x[R(x) \wedge \text{chase}'(x, r)\ ]$ (applied lambda Q)

Example 4, continued

4 `Vintrans -> barks`
$\lambda z[\text{bark}'(z)]$

5 barks and chases Rover

`VP[conj=no] ->`
`  VP1[conj=no] VP2[conj=yes]`

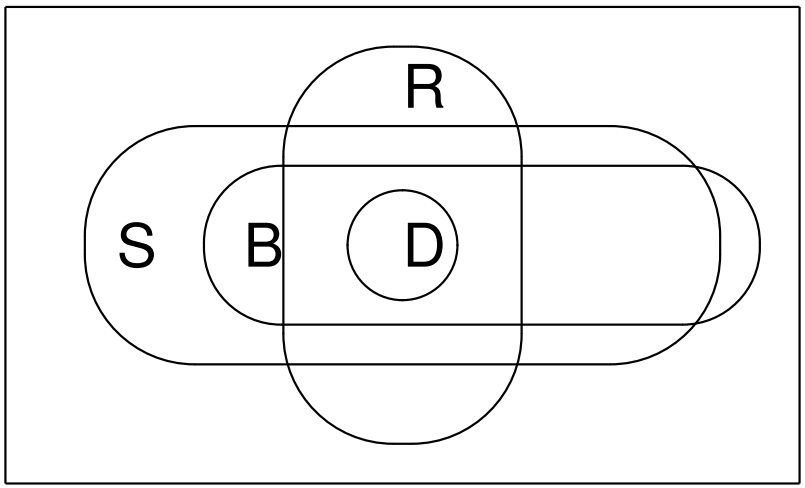
$\text{VP2}'(\text{VP1}')$ (grammar rule)
$\lambda R \lambda x[R(x) \wedge \text{chase}'(x, r)](\lambda z[\text{bark}'(z)])$ (sub. VP1, VP2)
$= \lambda x[\lambda z[\text{bark}'(z)](x) \wedge \text{chase}'(x, r)]$ (applied lambda R)
$= \lambda x[\text{bark}'(x) \wedge \text{chase}'(x, r)]$ (applied lambda z)

6 Kitty barks and chases Rover

`S -> NP VP`
$\text{VP}'(\text{NP}')$
$\lambda x[\text{bark}'(x) \wedge \text{chase}'(x, r)](k)$
$= \text{bark}'(k) \wedge \text{chase}'(k, r)$

Denotation and type of quantifiers

every dog denotes the set of all sets of which dog′ is a subset.
i.e., a function which takes a function from entities to truth values and returns a truth value.
For instance, *every dog* might denote the set $\{\text{bark}', \text{run}', \text{snore}'\}$:

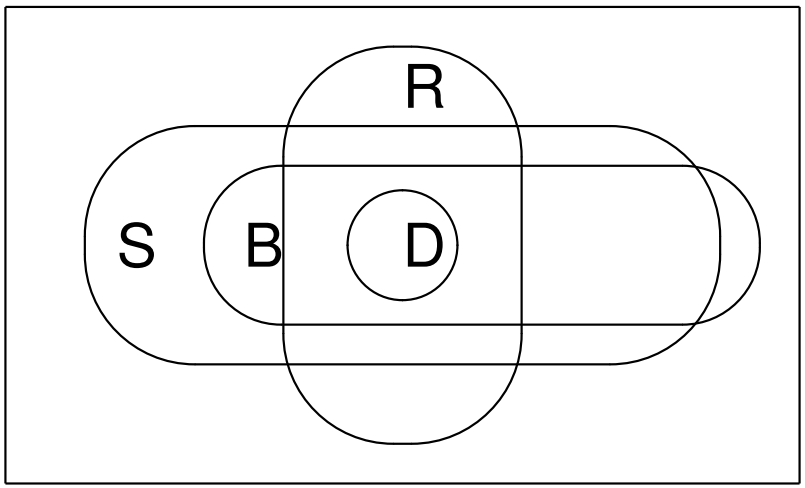


D= dog′, B= bark′, S= snore′, R= run′

What does *some dog* denote?

Denotation and type of quantifiers

every dog denotes the set of all sets of which dog′ is a subset.
i.e., a function which takes a function from entities to truth values and returns a truth value.
For instance, *every dog* might denote the set $\{\text{bark}', \text{run}', \text{snore}'\}$:



D= dog′, B= bark′, S= snore′, R= run′

What does *some dog* denote?

Denotation and type of quantifiers, continued

The type of *every dog* is $\langle\langle e, t\rangle, t\rangle$ (its argument has to be of the same type as an intransitive verb).
every dog:

$$\lambda P[\forall x[\text{dog}'(x) \implies P(x)]]$$

Semantically, *every dog* acts as a functor, with the intransitive verb as the argument:

$$\begin{aligned}
&\lambda P[\forall x[dog'(x) \implies P(x)]](\lambda y[sleep(y)]) \\
&= \forall x[dog'(x) \implies \lambda y[sleep(y)](x)] \\
&= \forall x[dog'(x) \implies sleep(x)]
\end{aligned}$$

This is higher-order: we need higher-order logic to express the FOPC composition rules. Problem: *every dog* acts as a functor, *Kitty* doesn't, so different semantics for `S -> NP VP`, depending on whether NP is a proper name or quantified NP.

Denotation and type of quantifiers, continued

The type of *every dog* is $\langle\langle e, t\rangle, t\rangle$ (its argument has to be of the same type as an intransitive verb).
every dog:

$$\lambda P[\forall x[\text{dog}'(x) \implies P(x)]]$$

Semantically, *every dog* acts as a functor, with the intransitive verb as the argument:

$$\begin{aligned}
&\lambda P[\forall x[dog'(x) \implies P(x)]](\lambda y[sleep(y)]) \\
&= \forall x[dog'(x) \implies \lambda y[sleep(y)](x)] \\
&= \forall x[dog'(x) \implies sleep(x)]
\end{aligned}$$

This is higher-order: we need higher-order logic to express the FOPC composition rules. Problem: *every dog* acts as a functor, *Kitty* doesn't, so different semantics for `S -> NP VP`, depending on whether NP is a proper name or quantified NP.

Type raising

Change the type of the proper name NP: instead of the simple expression of type e, we make it a function of type $\langle\langle e, t\rangle, t\rangle$
So instead of k we have $\lambda P[P(k)]$ for the semantics of *Kitty*.
But, what about transitive verbs? We've raised the type of NPs, so now transitive verbs won't work.
Type raise them too . . .
chases:
$\lambda R[\lambda y[R(\lambda x[chase(y, x)])]]$

Executive Summary:

- this gets complicated,
- and *every cat chased some dog* only produces one scope!

Type raising

Change the type of the proper name NP: instead of the simple expression of type e, we make it a function of type $\langle\langle e, t\rangle, t\rangle$
So instead of k we have $\lambda P[P(k)]$ for the semantics of *Kitty*.
But, what about transitive verbs? We've raised the type of NPs, so now transitive verbs won't work.
Type raise them too . . .
chases:
$\lambda R[\lambda y[R(\lambda x[chase(y, x)])]]$

Executive Summary:

- this gets complicated,
- and *every cat chased some dog* only produces one scope!

Computational compositional semantics

- Event variables: chase(e,x,y) etc
- Underspecification of quantifier scope
- Alternatives to lambda calculus for composition (sometimes)
- Robustness techniques
- Integration with distributional methods (very recent)