UNIB20005 Language & Computation
Analysing the meaning of sentences (2)

Lesley Stirling, School of Languages & Linguistics

The story so far ...

- Major assumptions in composing semantic representations:
  - The Principle of Compositionality
  - The ‘rule-to-rule hypothesis’
- Associating semantics with syntactic categories by means of the **feature** SEM=<>
- Inheritance of values of SEM from head daughter constituents to mother constituents
- Function application of the semantic value of the VP to the semantic value of the subject NP

The story so far ...

The process of semantic interpretation:

$$\begin{tikzcd}
\text{Natural Language sentence} \arrow[r] & \text{Logical translation} \arrow[r] & \text{Interpretation with respect to a model}
\end{tikzcd}$$

Cyril barks.

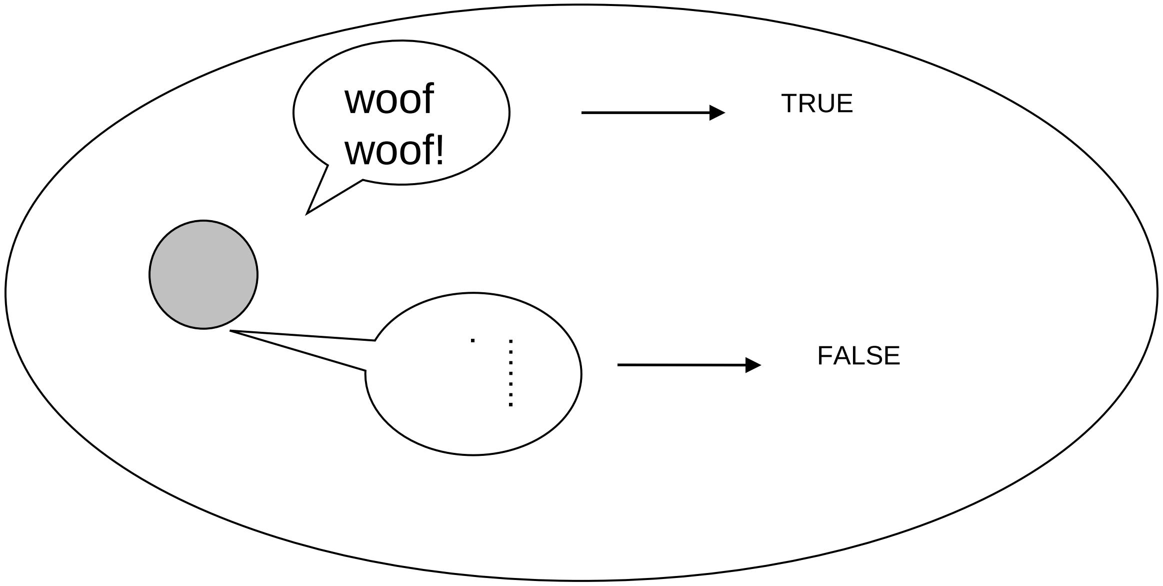
bark(cyril)

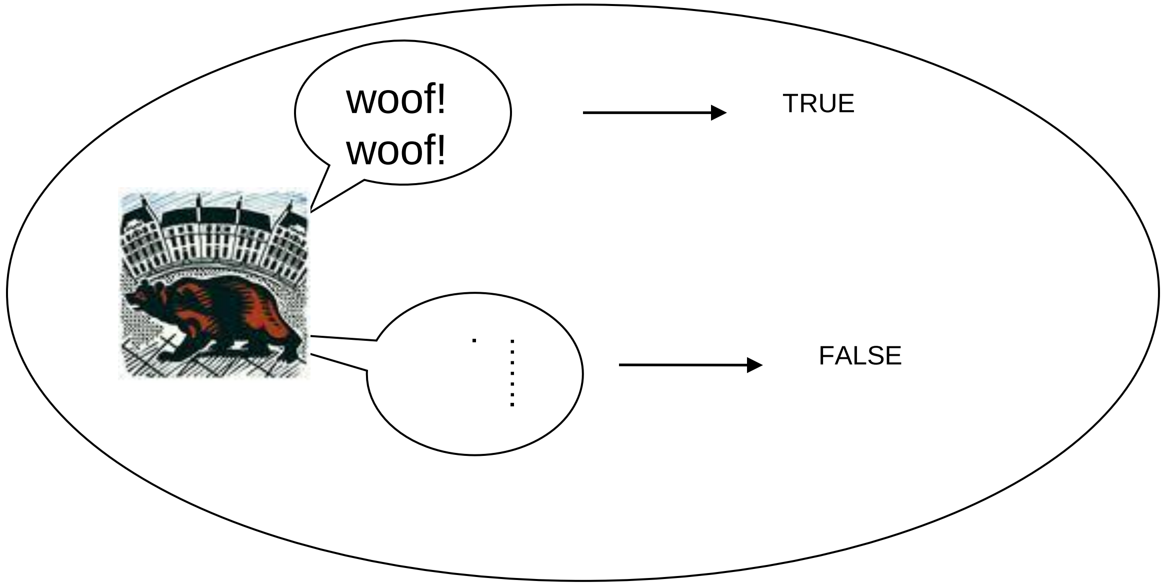
The story so far ...

- A model as a partial specification of some way one possible world might be
- The nature of the meanings of different kinds of expressions, and the concept of logical types:
  - Names (proper nouns) denote entities (type <e>)
  - Sentences denote truth values (type <t>)

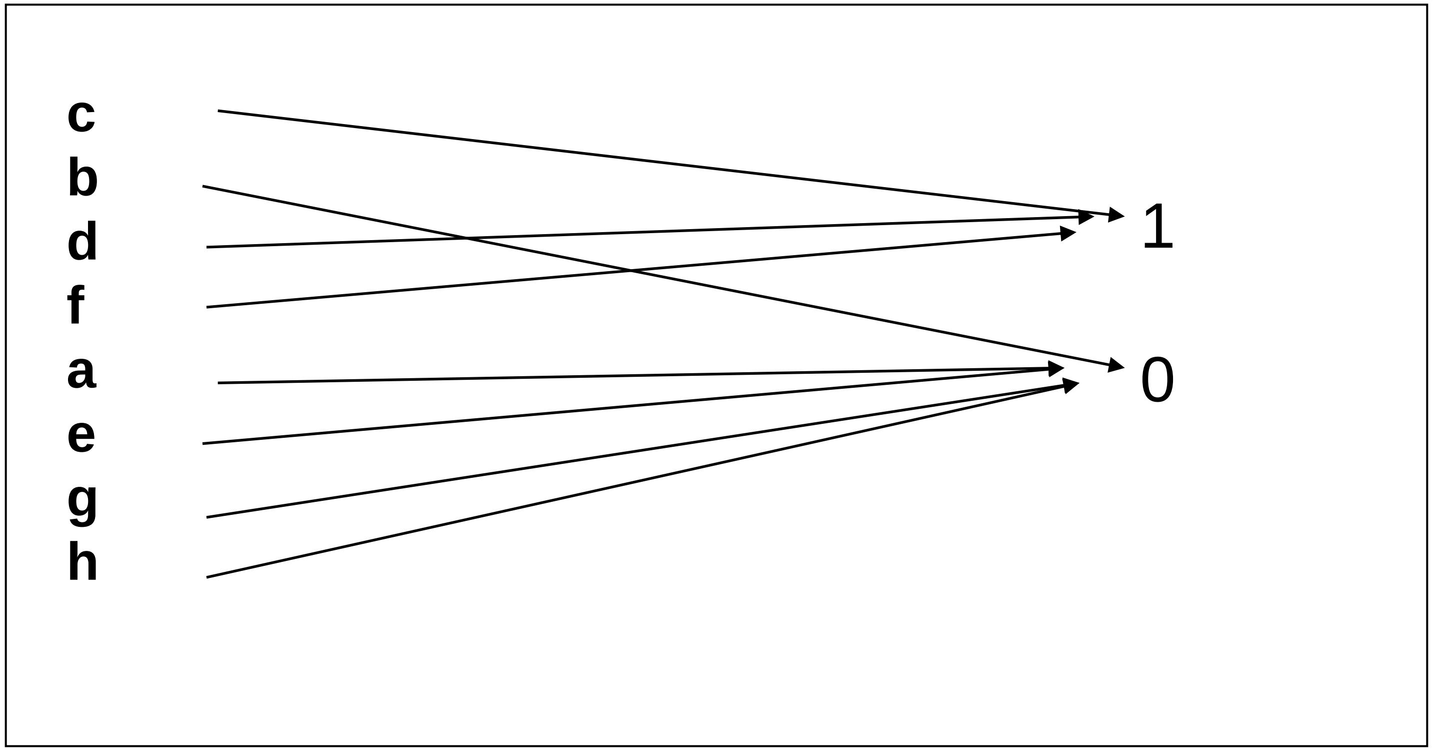
predicates

- Predicates denote properties
- the meaning of a one-place predicate like *bark* can be thought of as an **incomplete proposition**
- a predicate or rather the property it denotes is an **unsaturated proposition**


woof woof!
TRUE
. ⋮
FALSE


woof!
woof!
TRUE
FALSE

One-place predicates <e,t>
e.g. Val(bark)={c,d,f}



Worlds and models

- there are infinitely many *possible worlds* or *possible situations*
- the real world is a possible world
- **Models:** We can think of a model as a partial specification of some way one possible world might be

Let’s define a model *M1*.

In *M1*:
The logical translation of *Cyril* – that is, the individual constant **cyril**,

DENOTES

.... Or for convenience let’s say **c**

The logical translation of *barks* – the logical predicate we have written **bark**

DENOTES a set .... **{ c,d,f...}**

and our **model theory** will specify the **Truth-conditions for one-place predicates** something as follows: A formula IV(NP) is true with respect to a model iff the entity denoted by NP in the model is a member of the set of entities denoted by IV in the model. The formula is false otherwise.

- If an S has two daughters, one an NP and the other a VP (here IV), the meaning of the S equals the result of allowing the meaning of the NP to saturate the meaning of the VP (IV).
- If a node has two daughters, and the meaning of one of the daughters is a thing (individual) and the meaning of the other is a property, then the meaning of the mother equals the result of allowing the thing (individual) to saturate the property.

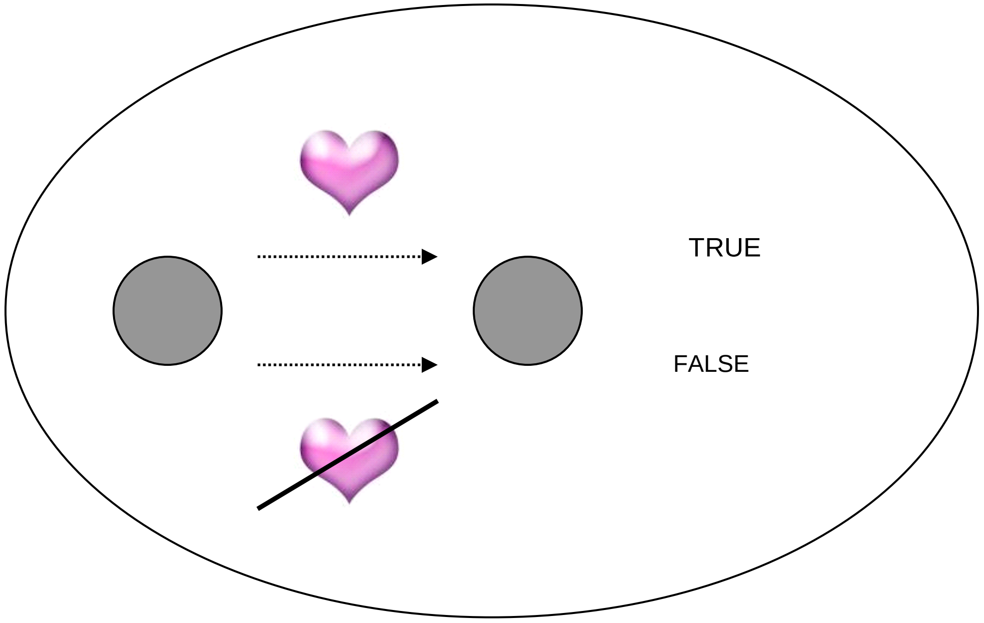
What about two-place predicates?

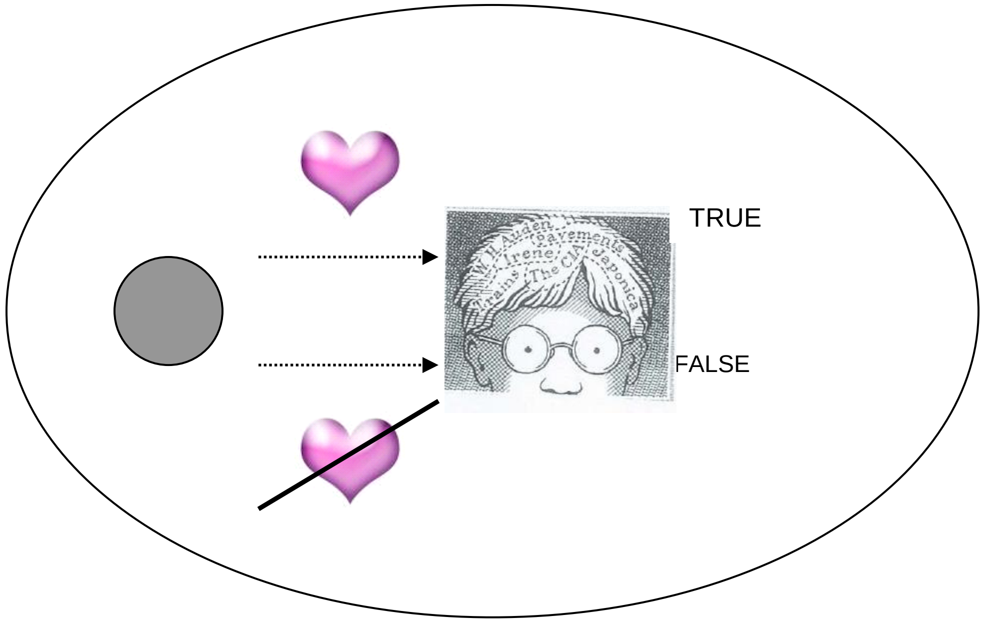
transitive verbs (two-place predicates)

e.g. *Cyril admired Bertie.*

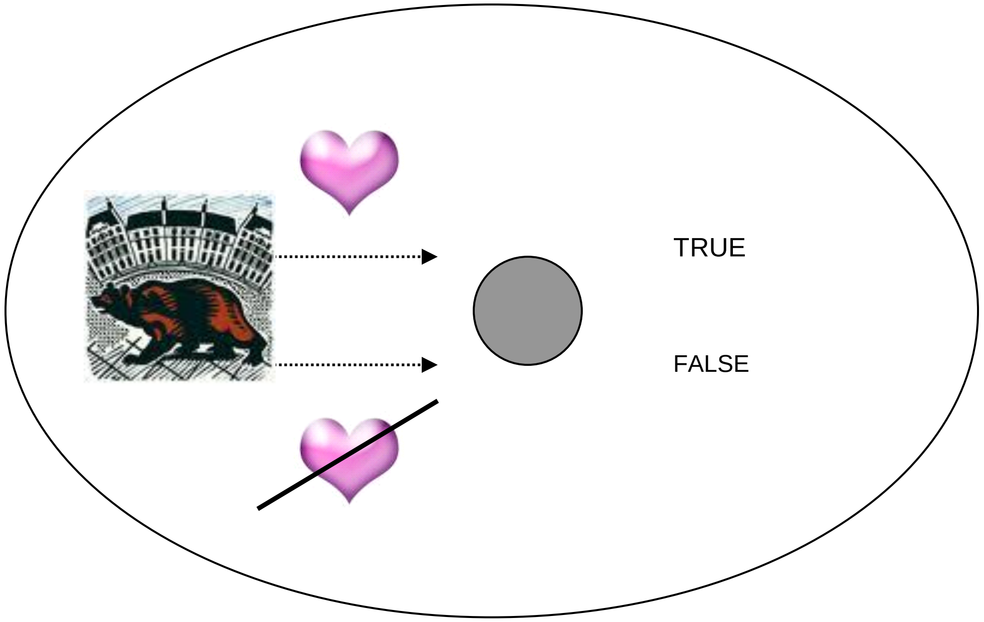
Logical translation: admire(cyril,bertie)

admire	*DENOTES*	***{<c,b>,***
		< c,a>,
		< a,c>,
		< b,c>}.


TRUE
FALSE


TRUE
FALSE

Relative clauses
Bertie is [who Cyril likes].



Thus the major components of a model are:

- The **Domain D** = the set of individuals in the model/world; these may in principle be of different kinds, **e.g.**
  - Entities
  - Times
  - Events

  …
- But for our purposes, just entities (type <e>)

- The **Denotation Assignment Function** Val = the function which assigns a denotation in the model to each basic expression in the logical translation**, e.g.**
  - For each individual constant c in the logical language *L*, Val(c) is an element of D
  - For each predicate P of arity $n \geq 0$, Val(P) is a function from D^n to {True, False} – equivalently, the denotation of the predicate is a set of individuals or ordered pairs built up from D

- Models also require a **Variable assignment function g** = this is what allows variables to be interpreted.

Recall:

Everybody admires someone.

all x.(person(x) → exists y.(person(y) & admire(x,y)))

exists y.(person(y) & all x.(person(x) → admire(x,y)))

AND

IV[SEM=< \x.bark(x)>] → 'barks'

- Models also require a **Variable assignment function g** = this is what allows variables to be interpreted.

Recall:

Everybody admires someone.

all x.(person(x) → exists y.(person(y) & admire(x,y)))

exists y.(person(y) & all x.(person(x) → admire(x,y)))

AND

IV[SEM=< \x.bark(x)>] → ‘barks’

So:

Expressions in *L* are interpreted with respect not only to a model M but also to a VAF g, and denotations of expressions are written accordingly:

$[\quad\quad]^{M,g}$

Back to lambda

Partee: “lambdas changed my life”

the lambda operator

A third way of specifying sets:

1. List the members {1,2,3,4,5,6,7,8,9}
2. Define a characteristic function for the set
3. Describe in general terms the **property** that picks out all the members of the set

   ‘the set of all x such that x is a natural number less than 10’

Intensional definitions of sets:

- Select a variable x and specify a property that any entity substituting for the variable must have to be a member of the set.

  {x | x is a natural number less than 10}

{x | x admired Bertie}

{x | Cyril bit x}

{z | z was crazy and z was a dog}

{x | (admire(bertie)) (x)}

{x | (bite(x)) (cyril)}

{x | crazy(z) & dog(z)}

But…

We can’t translate VPs as:

- The whole intensional definition (property description), e.g.
  {x | (kick(ethel)) (x)}
- The propositional function, e.g.
  (kick(ethel)) (x)

So …

We use a special logical operator to turn the propositional function into something which has type <e,t> and which denotes a characteristic function over sets.

This is the **lambda operator λ.**

λx [(admire(bertie)) (x)]

λx [(bite(x)) (cyril)]

λz [crazy(z) & dog(z)]

History:

- Devised by logician Alonzo Church
- In the 1930s
- The key element in his ‘lambda calculus’
- His motivation: to provide an unambiguous, compositional way to refer to functions

syntax of our logical language *L*:

[1] Recursive definition of types

a. e is a type,

b. t is a type,

c. If a is a type and b is a type, then <a,b> is a type.

d. If f is an expression of type t containing an unbound instance of a variable x of type e, then λx[f] is a well-formed expression of type <e,t>.

[2] Rule of functional application (RFA)

If *f* is an expression of type <a,b>, and *a* is an expression of type a, then *f(a)*is an expression of type b.

Lambda conversion (β-reduction):

λx [(bite(olive)) (x)]   (cyril)

‘Cyril is an x such that x bit Olive’

>> (bite(olive)) (cyril)

[or: bite(cyril,olive)]

An unambiguous way of referring to functions:

- Without the lambda operator, a function name can only be unambiguously given by a prose description: ‘Let f be the function from D to R such that $f(x) = x^2 + 3$’
- The expression ‘$f(x)$’ and the expression

  ‘$x^2 + 3$’ are both ambiguous between reference to the **function**, and to the **value** of the function given argument x.
- Using lambda, we can unambiguously refer to the function as: $\lambda x\ [x^2 + 3]$.
- Then the application of the function to a particular argument is written as: $\lambda x\ [x^2 + 3]\ (z)$ – equivalent to

  $z^2 + 3$

What makes λ so useful to (computational) linguists?

- Lambda expressions: give compositionally analyzable names to functions of arbitrary types.
- English has:
  - a comparatively large number of types of syntactic category,
  - numerous recursive mechanisms which result in the embedding of expressions of one category in expressions of another / the same category
- Lambda is useful for analyzing complex expressions which work logically like simple ones, but which we have to (wish to) give a compositional account for.
- We end up with the same overall translation, but the difference is whether it is arrived at compositionally on the basis of a reasonable natural language syntax or not.

- Introducing the lambda operator essentially gives us a way of forming whatever **functions** we want, with the input of the function being the **type** of the variable being abstracted over, and the output of the function being the **type** of the open sentence in the scope of the operator.
- Thus λ has the crucial property of always changing the type of the open expression it binds, for example from type **t** to a functional type, e.g. **<e,t>**.

[1] ‘Passive’ constructions

Active version:

Cyril bit Olive.

Passive version:

Olive was bitten by Cyril.

‘Passive’ constructions

- Translate **passive VPs** by a lambda expression that abstracts on a variable in the appropriate position:

  bitten by Cyril => λx [(bite(x))(cyril)]

  (which denotes the set of things Cyril bit)
- This expression is then functionally applied to the translation of the subject:

  Olive was bitten by Cyril.

  λx [(bite(x))(cyril)] (olive) >> (bite(olive))(cyril)

Note:

- As we’ve seen, we can have more than one lambda abstraction within an expression.
- Lambda operators can abstract not only over individual variables, as we have seen so far, but also over variables of other types, e.g. property variables or formula variables:
  - $\lambda p\ [\text{snow} \leftrightarrow p]$
  - $\lambda P\ [P(\text{the-cat})]$

[2] Relative clauses

- The interpretation of a relative clause can be seen as a function of the interpretation of the corresponding open sentence.
- 2 aspects to be accounted for:
  - The internal structure and semantics of the relative clause
  - The syntax and semantics of the NP within which the relative clause acts as a modifier

Relative clauses:

- Play an adjectival (modifying) role, but are formed from sentences
- However, these are sentences containing a gap
- Hence, lambda abstraction is an ideal way of combining them: we assume the relevant underlying sentences are open sentences containing a free variable in the gap ('relativized') position

Kim loves x.

who(m) Kim loves;

that Kim loves;

such that Kim loves him/her

(love(x)) (kim)

λx [(love(x)) (kim)]

the property of being an x such that Kim loves x

y hates Lee.

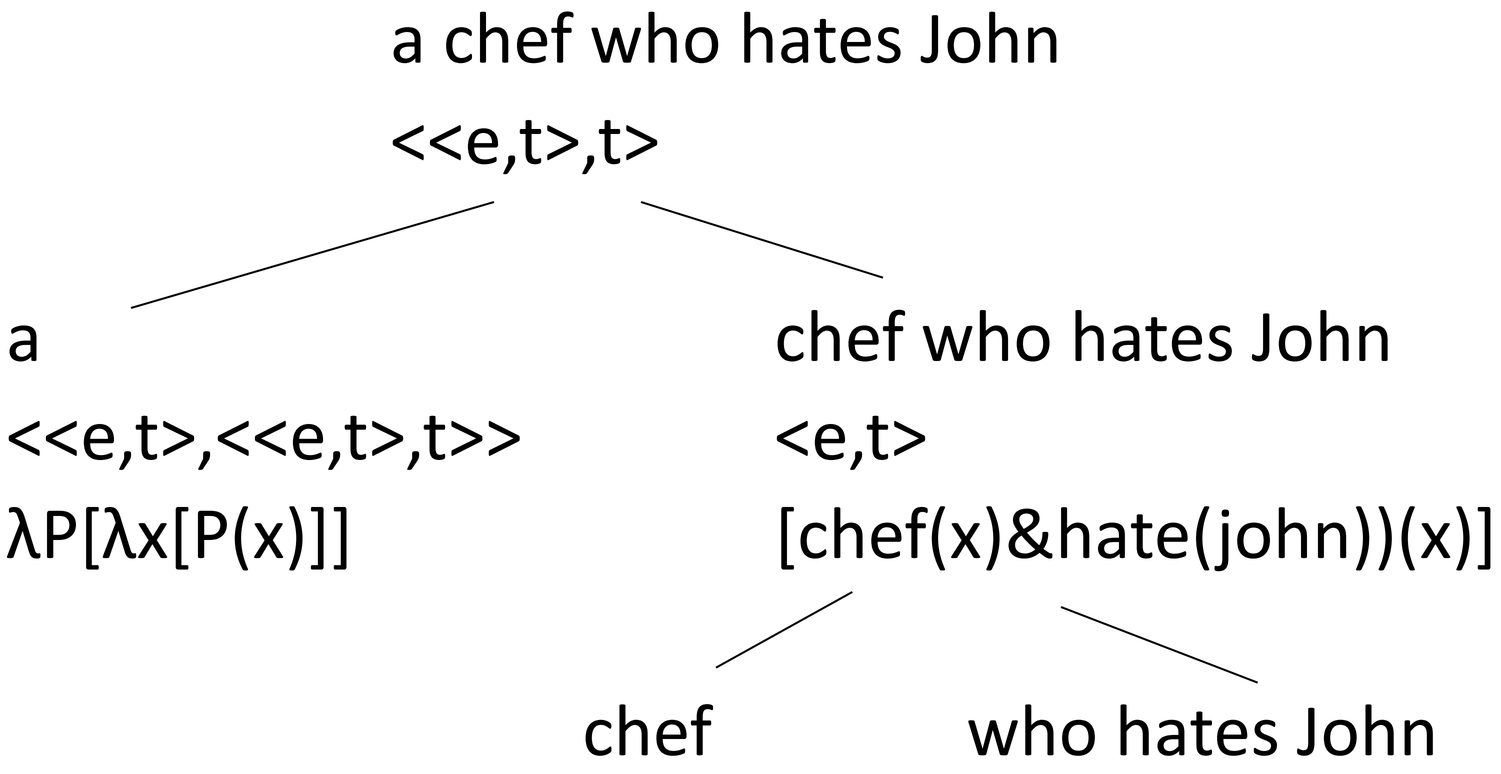
who hates Lee;

that hates Lee;

such that she/he hates Lee

(hate(lee)) (x)

λx [(hate(lee)) (x)]


a chef who hates John
<<e,t>,t>
a
<<e,t>,<<e,t>,t>>
λP[λx[P(x)]]
chef who hates John
<e,t>
[chef(x)&hate(john))(x)]
chef
who hates John

[3] Phrasal conjunction

- Lambda allows us to account for coordinate constructions
- Lambda allows us to relate:

Phrasal negation	to	sentential negation
Phrasal conjunction		sentential conjunction
Phrasal disjunction		sentential disjunction

(3) (a) Lee walks and talks.

(b) Lee walks and Lee talks.

(4) (a) Some woman walks and talks.

(b) Some woman walks and some woman talks.

Boolean phrasal conjunction used to be handled in linguistic theory by positing a ‘conjunction reduction’ transformation:

Lee walks and Lee talks.

But we can handle the relationship in the semantics.

[4] VP deletion

- In particular, the distinction between ‘strict identity’ and ‘sloppy identity’ readings of sentences involving VP deletion.
- **Kim believes she's sick and Lee does too.**

‘strict identity’ reading:

Kim believes Kim is sick, and Lee believes Kim is sick too.

‘sloppy identity’ reading:

Kim believes Kim is sick, and Lee believes Lee is sick.

Kim believes she's sick:

a. λx [(believe (sick(x))) (x)] (kim)

b. (believe (sick(kim))) (kim)

believe she's sick:

λx [(believe(sick (x))) (x)] = P1

λx [(believe(sick (kim))) (x)] = P2

Lee does too

P1 (lee)

P2 (lee)

Other phenomena:

- other constructions exhibiting **unbounded dependencies.**

(with lambda abstraction on the position corresponding to the gap/pronoun in the construction)

  - WH questions [Groenendijk & Stokhof 1989]
  - focus phenomena
  - the comparative construction [Cresswell 1976]
  - topicalization [Gazdar et al 1985]
  - pseudocleft construction [Partee 1984]

- **lexical rules** of various kinds (esp. those which like passive **manipulate the argument frames** of verbs or VPs and other predicates).
  - causative verbs
  - detransitivization
  - morphological reflexives
  - adjectival *un*- prefixation
  - object raising
  - 'dative movement'
  - the relation between *seek* and *try to find*
  - and between *want* and *want to have*
  - other semantically regular parts of derivational morphology

Most importantly: the interpretation of NPs

- Standard logical formulae don’t fit the syntax of English NP + VP structure
- NPs are not treated uniformly despite having the same distribution syntactically
- NPs are not given a continuous translation

What we need to do at minimum:

- Account for a wide(r) range of NP types
- Give an account of the internal syntactic and semantic structure of NPs (including attributive adjectives)
- Maintain our adherence to the principle of compositionality and the rule-to-rule hypothesis
- Give a different (more adequate?) account of proper names and definite NPs

[1] The traditional account of quantified NPs

Existential quantification

- someone/thing
- a book
- some cat
- at least one lecturer

Notation: ∃ **(exists)**

Universal quantification

- everyone/thing
- every dog
- each person
- all lecturers

Notation: ∀ **(all)**

Gottlob Frege 1848-1925

"Modern logic stands in contrast to all the great logical systems of the past … in being able to give an account of sentences involving multiple generality, an account which depends upon the mechanism of quantifiers and bound variables…"

Michael Dummett (1973) *Frege: Philosophy of Language*

∀and ∃

- Like λ, they bind variables

λx [bark(x)]	type <e,t>
∀x [bark(x)]	type <t>
∃x [bark(x)]	type <t>

- Unlike λ, they don’t change the type of the open expression within their scope.
- Like λ, they are interpreted via the VAF g.

Interpretation (informally)…

Everyone laughed. => $\forall x$ [laugh(x)]

To establish the truth or falsity of the formula, intuitively, you go through every possible assignment of a value to the variable x from the model, and check whether this entity is part of the denotation of laugh. If every such assignment is part of the denotation of the predicate, then the universally quantified sentence is true with respect to that model.

Informal motivation for the new account

Consider:

$\forall x\, [(crazy(x)\ \&\ student(x)) \rightarrow laugh(x)]$

(Every crazy student laughed.)

Intuitively:

This specifies a range of properties (**crazy**, **student**) and says that the further property predicated in the sentence (**laugh**) holds of anything which has these properties. (Anything which has the property of being crazy and being a student also has the property of laughing.)

This sentence can be seen as saying that the property of laughing has the higher order property of being true of every crazy student.

The account:

Every student laughed.

$\forall x$ [student(x) $\rightarrow$ laugh(x)]

This is a **formula** of type **t**.

The information in the subject NP *every student* (where the predicate variable P replaces the VP):

$\forall x$ [student(x) $\rightarrow$ P(x)]

This is still of type **t**, but now an **open sentence**. We can use **Lambda Abstraction** to bind the variable and allow the expression to be assigned a denotation:

$\lambda P\ \forall x$ [student(x) $\rightarrow$ P(x)]

We thus have an alternative type and translation for NPs.

Why we have to do this

- Everyone liked bertie.
- bertie
- (like(bertie))(x)
- we require the translation of *everyone* to combine with this to give the translation for the whole sentence:

  $\forall x$ [(like(bertie))(x)]
- There are 2 possible ways to combine the meanings of VP & NP:
  - As usual VP(NP) <e,t> and e
  - NP(VP) <e,t> and <<e,t>,t>
- We HAVE to take the second option, because we can't translate the quantifier pronoun into an expression of type e and get the right translation.
- The translation of *everyone* must be: $\lambda P\ [\forall x\ [P(x)]]$

Reflection

- Are we any closer to allowing natural language interrogation of databases using ordinary language questions? (Recall that part of the motivation for exploring a proper semantic theory was the lack of generality with the database queries first discussed in 10.1).
- Is it within the realms of possibility to use these techniques to write a program that would pass the Turing test?

To do

- Read Chapter 10.4 – ***esp. pp 386-394***
- Extra reading: Partee paper and Portner chapters are available from LMS
- Worksheet on semantics

Terminology from 10.3 on First-Order Logic

- Predicates (arity of predicates: unary, binary)
- Arguments
- Individual constants
- Individual variables
- Types (basic, complex)
- Existential quantifier ($\exists$ or exists)
- Universal quantifier ($\forall$ or all)
- Binding of variables (scope, free, bound)
- Open and closed formulae
- *See the summary of First-order logic on p 376*