

Computational Semantics

Adding Natural Language

Adding Natural Language

- ◆ Task for today: Translating NL input into first order logic
 - Subtasks:
 - ◆ Parsing NL sentences
 - Definite Clause Grammars
 - ◆ Rule by rule translation
 - Lambdas calculus
 - Beta reduction

Syntax/Semantics relation

◆ Assumptions

- Syntax of a language determining what the grammatical structures are
- Semantic theory interprets those structures

◆ Adopt Montague's assumption

- "there is no theoretically important distinction between formal and natural languages"

Semantic Composition

◆ Frege's principle:

- The meaning of the whole is a function of the meaning of their parts and their mode of composition
 - ◆ We need to know what the parts are (and how they are put together). That is the job of syntax!
 - ◆ Then we need to know what kind a of functions meanings are

Syntax

◆ What are the parts of NL sentences?

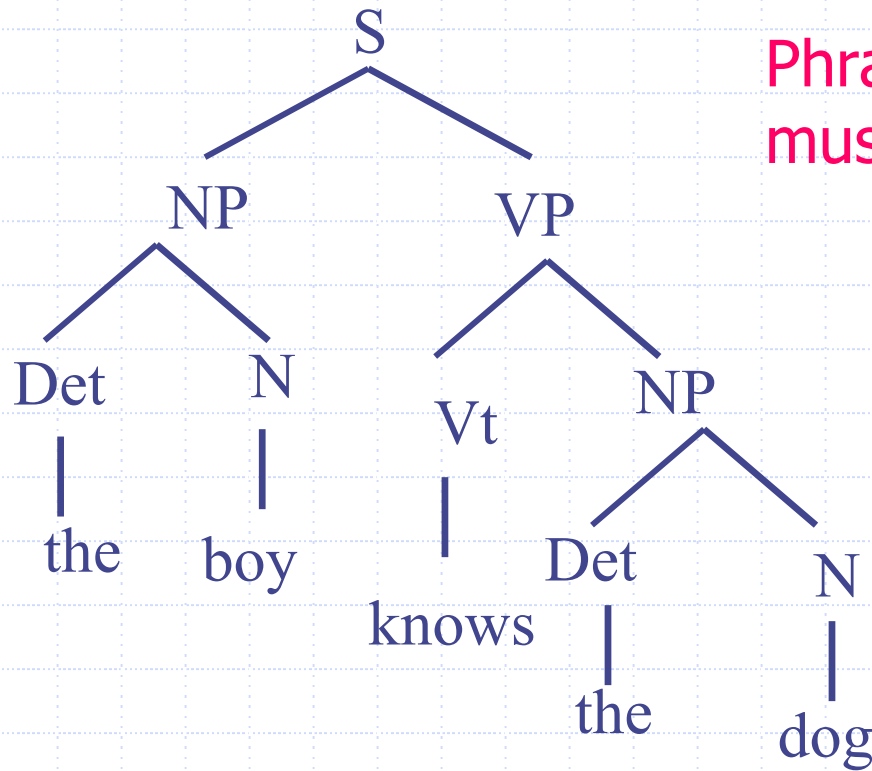
■ Simple parts:

- ◆ Words/Morphemes/Lexical items
"John", "love", "have", etc.

■ Complex parts:

- ◆ Phrases etc.

Phrase Structure



Phrase structure
must be rule generated

Simple parts
can be listed

Phrase Structure Syntax

◆ Build phrases via rules.

S --> NP VP

NP --> Det N

VP --> Vt NP

VP --> Vi

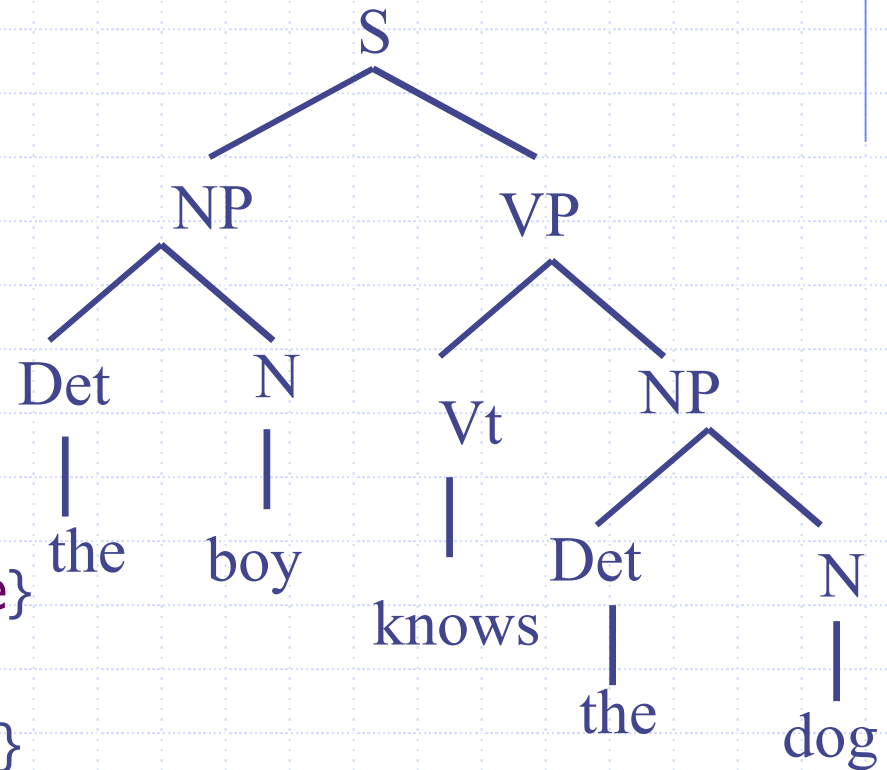
NP --> {John, Mary}

Det --> {a, the, every, some}

N --> {boy, girl, dog}

Vt --> {loves, hates, knows}

Vi --> {smiles, laughs}



Beyond Phrase Structure

◆ Agreement phenomena

- Subject/Verb agreement
The boy likes the girl.
The boys like the girl.

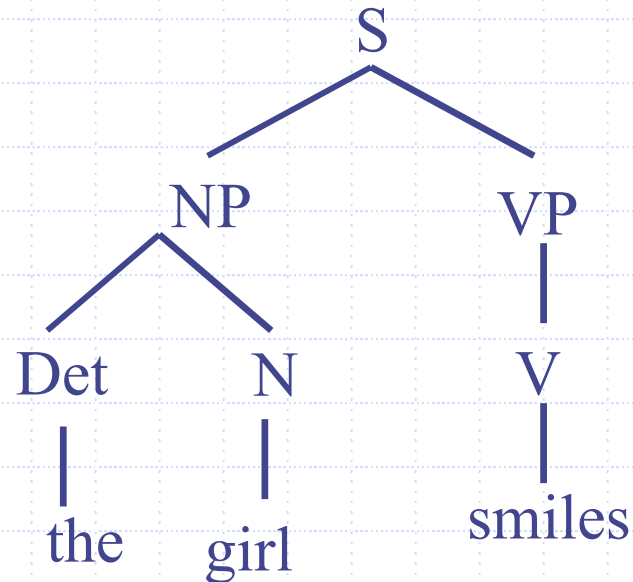
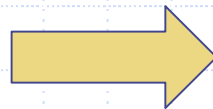
◆ Long Distance dependency

- Tag Questions:
Peter didn't leave, did he?
Peter wouldn't leave, would he?
Peter told Mary that he would leave, didn't he?
- "Movement"
Who did Peter tell Mary he would invite to the party?
The boy who Peter told Mary he would invite to the party left.

Parsing

◆ What is parsing

- Assignment of structure to sequence of simple elements
- "the" "girl" "smiles"



Parsing in Prolog

- ◆ A sentence is input as a list of atoms
 - `[the, girl, smiles]`
- ◆ Parsing = proving that this list is generated by the grammatical rules of the language
 - ? `s([the, girl, smiles]).`
- ◆ PS rules can be written as Prolog clauses
 - `(consult clausGram.pl)`

Good Parsing in Prolog

- ◆ A more common (and more efficient) Prolog parser is implemented so:

```
s(L1,L) :- np(L1,L2),vp(L2,L).
np(L1,L) :- det(L1,L2), n(L2,L).
vp(L1,L) :- vt(L1,L2), np(L2,L).
vp(L) :- vi(L).
det([the|L],L).
det([a|L],L).
n([man|L],L).
n([girl|L],L).
vt([loves|L],L).
vi([smiles|L],L).
```

- Consult `clausGram2.pl`

Built-in DCG reading

◆ Prolog reads

```
s --> np, vp.  
np --> det, n.  
vp --> vt, np.  
n --> [man]
```

As

```
s(L1,L) :- np(L1,L2),vp(L2,L).  
np(L1,L) :- det(L1,L2), n(L2,L).  
vp(L1,L) :- vt(L1,L2), np(L2,L).  
n([man|L],L).
```

Grammar Engineering

◆ We can simply write DCG rules

- Parsing follows via queries like
`s([john, loves, Mary], []).`
- And Generation via
`s(Sentenc, []).`

◆ Consult `DCG.pl`

Grammar Engineering

◆ Rules can have extra features

`s --> np[Agr], vp[Agr].`

`np[Agr] --> det, n[Agr].`

are read as

`s(L1,L) :-
 np(Agr,L1,L2), vp(Agr,L2,L).`

`np(Agr,L1,L) :- det(L1,L2),
 n(Agr,L2,L).`

Grammar Engineering

- ◆ Rules can also have extra conditions. These are added between “{” and “}”.

```
s --> np[AgrN], vp[AgrV],  
      {checkAgr(AgrN,AgrV)}.
```

is read as

```
s(L1,L) :-  
  np(AgrN,L1,L2),  
  vp(AgrV,L2,L),  
  checkAgr(AgrN,AgrV).
```

- ◆ Consult `DCGfeat.pl`

Getting the Syntax Out

◆ How to return an analysis?

- We will can use a feature.

```
s(s(SynNP, SynVP)) -->  
  np(SynNP), vp(SynVP).
```

```
vp(vp(SynV, SynNP)) -->  
  vt(SynV), np(SynNP).
```

```
np(np(SynD, SynN)) ---->  
  det(SynD), n(SynN).
```

◆ Consult DCGfeatSyn.pl

More Complexity

- ◆ In `DCGfeatSynComb.pl` we combine agreement and syntactic tree construction.

```
s([syn:s(Syn1, Syn2)]) -->
    np([agr:Agr, syn:Syn1]),
    vp([agr:Agr, syn:Syn2]).

np([agr:Agr, syn:np(Syn1, Syn2)]) -->
    det([agr:Agr, syn:Syn1]),
    n([agr:Agr, syn:Syn2]).

vp([agr:Agr, syn:vp(Syn1, Syn2)]) -->
    vt([agr:Agr, syn:Syn1]),
    np([agr:_, syn:Syn2]).

vp(X) --> vi(X).

np([agr:sg, syn:np(john)]) --> [john].
np([agr:sg, syn:np(mary)]) --> [mary].
```

Semantic Analysis

But what we want is a semantic analysis!

- ◆ Adopt the “rule by rule” hypothesis -
 - For every syntactic rule there is a semantic combination part:

```
s[ syn: Syn, sem: Sem ] -->
  np[ syn: SynNP, sem: SemNP ],
  vp[ syn: SynVP, sem: SemVP ],
  { combineSemantics( SemNP, SemVP, Sem ) }.
```

- What should `combineSemantics` do?

Semantic Combination

◆ We want to get

```
Sem = love(john,mary)
```

out of

```
s(Sem,[john, loves, mary],[ ])
```

◆ How do we do it?

- Naïve approach: Argument substitution (naiveSem.pl)

```
np(mary) --> [mary].
```

```
np(john) --> [john].
```

```
v(love(X,Y)) --> [love].
```

```
s(Sem) --> np(SemNP), vp(Sem),  
  {arg(1,Sem,SemNP)}.
```

```
vp(Sem) --> v(Sem), np(SemNP),  
  {arg(2,SemVP,SemNP)}.
```

The Problem: Quantifiers

How can we treat: "John loves every girl"

- We do NOT want:
 - ◆ `love(john, every girl)`
 - ◆ `love(john, every(x, girl(x)))`
 - ◆ `love(john, every(girl(x), Y))`
- We want:
 - ◆ `all(x, imp(girl(x), love(john, x)))`
 $\forall x[\text{girl}(x) \rightarrow \text{love}(\text{john}, x)]$
- But where is "every girl" in that formula?
 $\forall x[\text{girl}(x) \rightarrow _(_, x)]$

Functional Combination

Remember Frege?

- Semantics of the whole is a function of the semantics of the parts
 - ◆ We can think of semantic combination as function application
 - ◆ Functions

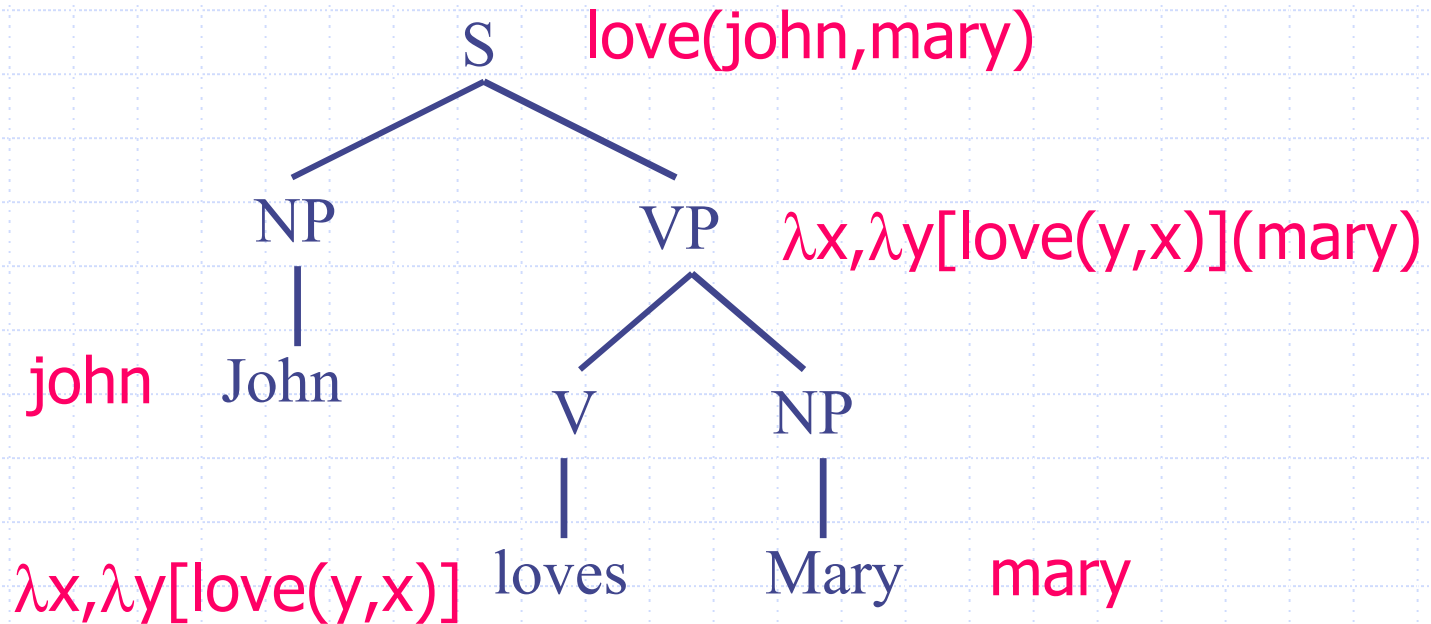
How do we decide what the meaning of “every girl” is?

Practical method: we build a function that, when combined with the rest of the sentence gives us what we want.

Basis of Formal Semantics since Montague (1970)

Compositional Interpretation

◆ The technicalities



Lambda Notation

The λ is a variable binder (like a quantifier), that works as a “slot filler”

$$\lambda x [\text{happy}(x)](\text{peter}) = \text{happy}(\text{peter})$$

$$\lambda x [\text{like}(x,x)](\text{peter}) = \text{like}(\text{peter},\text{peter})$$

$$\lambda x \lambda y [\text{like}(x,y)](\text{peter}) = \lambda y [\text{like}(\text{peter},y)]$$

$$\lambda y [\text{like}(\text{peter},y)](\text{mary}) = \text{like}(\text{peter},\text{mary})$$

$$\lambda P [P(\text{john})](\text{happy}) = \text{happy}(\text{john})$$

β conversion

Lambda Notation

β -conversion only λ -bound variables

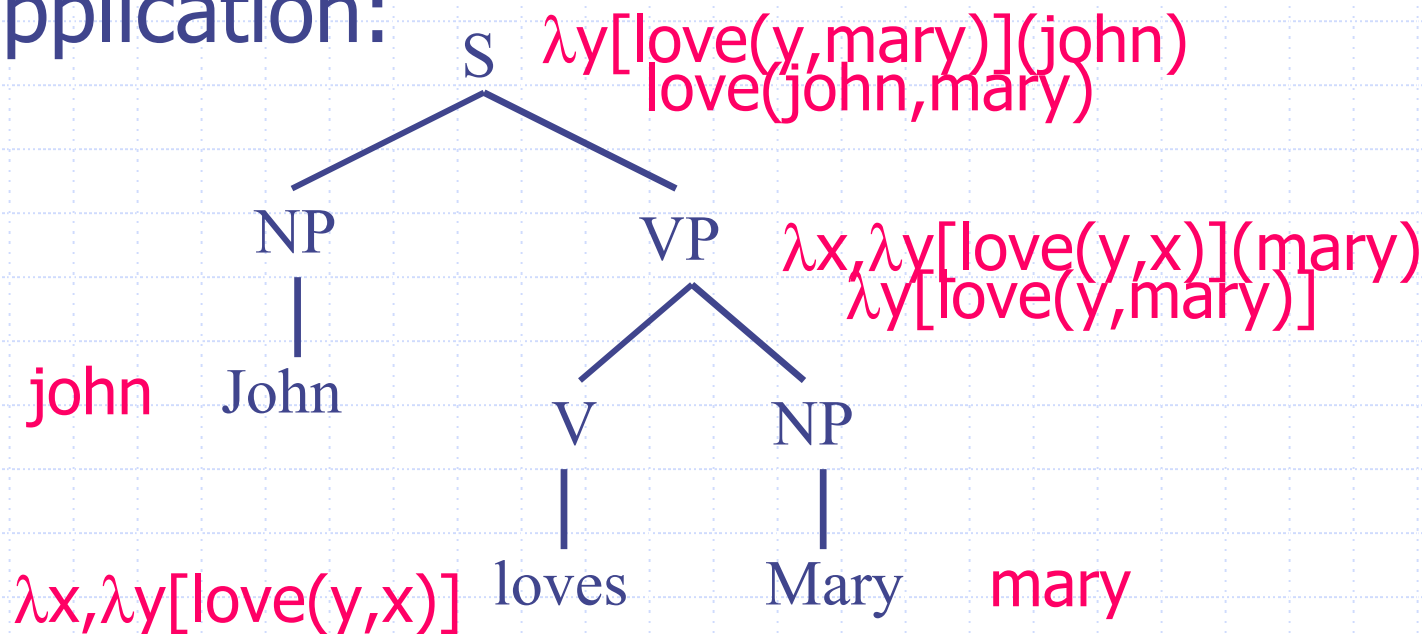
$$\lambda x [\text{happy}(x) \ \& \ \forall x [\text{man}(x) \rightarrow \text{happy}(x)]](\text{peter}) \\ = \text{happy}(\text{peter}) \ \& \ \forall x [\text{man}(x) \rightarrow \text{happy}(x)]$$

$$\lambda P \ \lambda Q \ \lambda x [P(x) \ \& \ Q(y)](\text{happy})(\text{tired})(\text{john}) = \\ \lambda Q \ \lambda x [\text{happy}(x) \ \& \ Q(y)](\text{tired})(\text{john}) = \\ \lambda x [\text{happy}(x) \ \& \ \text{tired}(y)](\text{john}) = \\ [\text{happy}(\text{john}) \ \& \ \text{tired}(\text{john})]$$

What might $\lambda P \ \lambda Q \ \lambda x [P(x) \ \& \ Q(y)]$ be the meaning of?

Compositional Interpretation

- ◆ To combine meanings we do function application:



Quantifier Interpretation

$\forall x[\text{girl}(x) \rightarrow \text{snorts}(x)]$

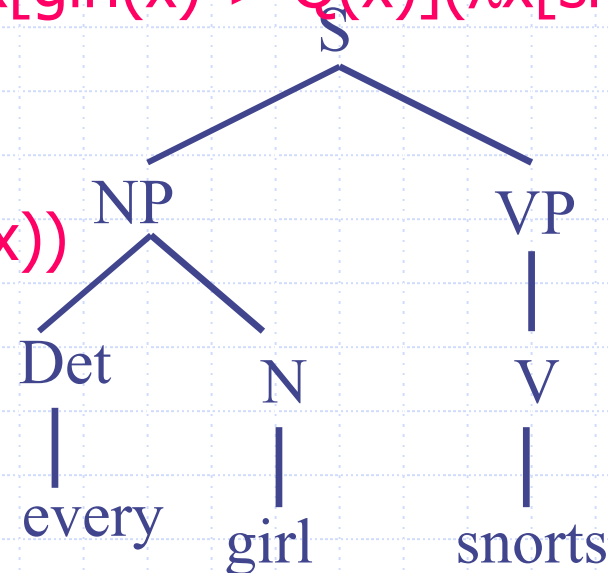
$\forall x[\text{girl}(x) \rightarrow \lambda x[\text{snorts}(x)](x)]$

$\lambda Q, \forall x[\text{girl}(x) \rightarrow Q(x)](\lambda x[\text{snorts}(x)])$

$\lambda Q, \forall x[\text{girl}(x) \rightarrow Q(x)]$

$\lambda Q, \forall x[\lambda x[\text{girl}(x)](x) \rightarrow Q(x)]$

$\lambda P, \lambda Q, \forall x[P(x) \rightarrow Q(x)](\lambda x \text{ girl}(x))$



$\lambda P, \lambda Q, \forall x[P(x) \rightarrow Q(x)]$

$\lambda x[\text{girl}(x)]$

$\lambda x[\text{snorts}(x)]$

What are lambda terms?

- ◆ They are functions, of course.
 - $\lambda x[P(x)]$
 - ◆ function from objects to truth values
 - $\lambda P [P(\text{john})]$
 - ◆ function from predicates to truth values.
 - $\lambda P \lambda x [P(x)]$
 - ◆ function from predicates to functions from objects to truth values.

Lambda terms in Prolog

Two parts:

- Lambda term
 - ◆ `lam(X, Formula)`
- Application $\lambda x[P(x)](\text{john})$
 - ◆ `app(lam(X, p(X)), john)`

◆ Consult file

`BlackburnJos/betaConvert.pl`

Ambiguity

- ◆ Multiple meanings associated with single sequence of elements
 - Lexical Ambiguity
 - ◆ Bank, duck, etc.
 - Structural Ambiguity
 - “Peter landed an airplane in a field.”
 - “Peter bought an airplane in a field.”

Structural Ambiguity

