

# EDAN20

## Language Technology

<http://cs.lth.se/edan20/>  
Chapter 14: Semantics and Predicate Logic

Pierre Nugues

Lund University  
[Pierre.Nugues@cs.lth.se](mailto:Pierre.Nugues@cs.lth.se)  
[http://cs.lth.se/pierre\\_nugues/](http://cs.lth.se/pierre_nugues/)

September 28, 2015



# The State of Affairs

Two people at a table, Pierre and Socrates, and a robot waiter.



# Formal Semantics

Its goal is to:

- Represent the state of affairs.
- Translate phrases or sentences such as *The robot brought the meal or the meal on the table* into logic formulas
- Solve references: Link words to real entities
- Reason about the world and the sentences.

A way to represent things and relations is to use first-order predicate calculus (FOPC) and predicate–argument structures



# Predicates

## Constants:

```
% The people:  
  'Socrates'.  
  'Pierre'.  
  
% The chairs:  
  chair1.      % chair #1  
  chair2.      % chair #2  
  
% The unique table:  
  table1.      % table #1
```

## Predicates to encode properties:

```
person('Pierre').  
person('Socrates').  
  
object(table1).  
object(chair1).  
object(chair2).  
  
chair(chair1).  
chair(chair2).  
table(table1).
```

## Predicates to encode relations:

```
in_front_of(chair1, table1).  
on('Pierre', table1).
```

# Prolog

Prolog is a natural tool to do first-order predicate calculus

- Things, either real or abstract, are mapped onto constants – or atoms: 'Socrates', 'Pierre', chair1, chair2.
- Predicates can encode properties: `person('Pierre')`, `person('Socrates')`, `object(table1)`, `object(chair1)`.
- Predicates can encode relations: `in_front_of(chair1, table1)`, `on('Pierre', table1)`.
- Variables unify with objects



# Querying the State of Affairs

Constants:

```
?- table(chair1).  
false.  
?- chair(chair2).  
true.
```

Variables:

```
?- chair(X).  
X = chair1;  
X = chair2
```

Conjunctions:

```
?- chair(X), in_front_of(X, Y), table(Y).  
X = chair1, Y = table1
```



# Logical Forms

Logical forms map sentences onto predicate-argument structures

*I would like to book a late flight to Boston*

```
would(like_to(i,  
  book(i,  
    np_pp(a(late(flight)),  
      X~to(X, boston)))))
```



# Compositionality

The principle of compositionality assumes that a sentence's meaning depends on the meaning its phrases

“The meaning of the whole is a function of the meaning of its parts.”

A complementary assumption is that phrases carrying meaning can be mapped onto constituents – syntactic units.

The principle of compositionality ties syntax and semantics together.

We saw that a predicate-argument structure could represent a sentence – the whole. How to represent the parts – the constituents?





# $\lambda$ -Calculus

The  $\lambda$ -calculus is a device to abstract properties or relations.

$$\lambda x.property(x)$$

or

$$\lambda y.\lambda x.relation(x,y)$$

A  $\lambda$ -expression is incomplete until a value has been given to it.  
Supplying such a value is called a  $\beta$ -reduction.

$$(\lambda x.property(x))entity\#1$$

yields

$$property(entity\#1)$$

In Prolog,  $X^{\wedge}property(X)$  represents  $\lambda x.property(x)$



# Nouns

Proper nouns: *Mark, Nathalie, Ludwig*

Common nouns (properties): *lecturer, book*:

$$\lambda x. \text{lecturer}(x) \quad \lambda x. \text{lecturer}(x)(\text{Bill}) = \text{lecturer}(\text{Bill})$$

Adjectives

$$\lambda x. \text{big}(x) \quad \lambda x. \text{big}(x)(\text{Bill}) = \text{big}(\text{Bill})$$

Adjectives and nouns: *big table*

$$\lambda x. (\text{big}(x) \wedge \text{table}(x))$$

Noun compounds are difficult: *lecture room*

$$\lambda x. (\text{lecture}(x) \wedge \text{room}(x)) \quad ?? \text{ Wrong!}$$

A better form is:

$$\lambda x. (\text{modify}(x, \text{lecture}) \wedge \text{room}(x))$$

although not completely satisfying



# Verbs

Verbs of being are similar to adjectives or nouns

Intransitive verbs	$\lambda x.rushed(x)$ $\lambda x.rushed(x)(Bill) = rushed(Bill)$
Transitive verbs	$\lambda y.\lambda x.ordered(x,y)$
Prepositions	$\lambda y.\lambda x.to(x,y)$



# Determiners

---

*A caterpillar is eating*

$\exists x, \text{caterpillar}(x) \wedge \text{eating}(x)$ , or  
 $\text{exists}(X, \text{caterpillar}(X), \text{eating}(X))$

---

*Every caterpillar is eating*

$\forall x, \text{caterpillar}(x) \Rightarrow \text{eating}(x)$ , or  
 $\text{all}(X, \text{caterpillar}(X), \text{eating}(X))$

---

*A caterpillar is eating a hedgehog*

$\exists x, \text{caterpillar}(x) \wedge (\exists y, \text{hedgehog}(y) \wedge \text{eating}(x, y))$ , or  
 $\text{exists}(X, \text{caterpillar}(X), \text{exists}(Y, \text{hedgehog}(Y), \text{eating}(X, Y)))$

---

*Every caterpillar is eating a hedgehog*

$\forall x, \text{caterpillar}(x) \Rightarrow (\exists y, \text{hedgehog}(y) \wedge \text{eating}(x, y))$ , or  
 $\text{all}(X, \text{caterpillar}(X), \text{exists}(Y, \text{hedgehog}(Y), \text{eating}(X, Y)))$

---

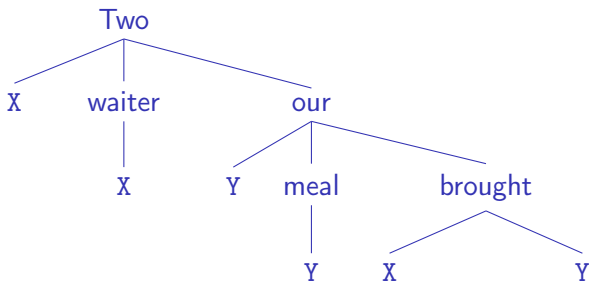


# Determiners: An Example

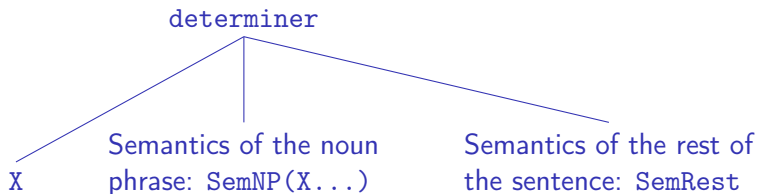
*Two waiters brought our meals*

translated into

`two(X, waiter(X), our(Y, meal(Y), brought(X, Y)))`



# General Representation of Determiners



Representation:

$(X \wedge NP) \wedge (X \wedge Rest) \wedge a(X, NP, Rest)$



# $\lambda$ -Representations

The partial, intermediate representations of  
*The waiter brought the meal* are

<i>waiter</i>	$X^{\text{waiter}}(X)$
<i>The waiter</i>	$(X^{\text{Rest}})^{\text{the}}(X, \text{waiter}(X), \text{Rest})$
<i>brought</i>	$Y^{\text{X}^{\text{brought}}}(X, Y)$
<i>meal</i>	$Y^{\text{meal}}(Y)$
<i>the meal</i>	$(Y^{\text{Verb}})^{\text{the}}(Y, \text{meal}(Y), \text{Verb})$

The operation to compose *brought the meal* is more complex It should produce something like:

$X^{\text{the}}(Y, \text{meal}(Y), \text{brought}(X, Y))$



# $\lambda$ -Representations (II)

We parse the verb phrase *brought the meal* using the rule

$\text{vp}(\text{SemVP}) \rightarrow \text{verb}(\text{SemVerb}), \text{np}(\text{SemNP}).$

We have:

$\text{SemVerb} = Y^X \text{brought}(X, Y)$

$\text{SemNP} = (Y^{\text{Verb}})^{\text{the}}(Y, \text{meal}(Y), \text{Verb})$

We just write the unification:  $\text{Verb} = \text{brought}(X, Y)$

Prolog returns:

?-  $\text{SemVerb} = Y^X \text{brought}(X, Y),$   
      $\text{SemNP} = (Y^{\text{Verb}})^{\text{the}}(Y, \text{meal}(Y), \text{Verb}),$   
      $\text{Verb} = \text{brought}(X, Y).$

$\text{SemVerb} = Y^X \text{brought}(X, Y),$   
 $\text{SemNP} = (Y^{\text{brought}(X, Y)})^{\text{the}}(Y, \text{meal}(Y), \text{brought}(X, Y)),$   
 $\text{Verb} = \text{brought}(X, Y).$





# Compositionality: The Lexicon

```
noun(X^waiter(X)) --> [waiter].  
noun(X^patron(X)) --> [patron].  
noun(X^meal(X)) --> [meal].  
verb(X^rushed(X)) --> [rushed].  
verb(Y^X^ordered(X, Y)) --> [ordered].  
verb(Y^X^brought(X, Y)) --> [brought].  
determiner((X^NP)^(X^Rest)^a(X, NP, Rest)) --> [a].  
determiner((X^NP)^(X^Rest)^the(X, NP, Rest)) --> [the].
```



# Interleaving Syntax and Semantics

```
s(Semantics) --> np((X^Rest)^Semantics), vp(X^Rest).
```

```
np((X^Rest)^SemDet) -->
```

```
    determiner((X^NP)^(X^Rest)^SemDet),
```

```
    noun(X^NP).
```

```
vp(Subject^Verb) --> verb(Subject^Verb).
```

```
vp(Subject^Predicate) -->
```

```
    verb(Object^Subject^Verb),
```

```
    np((Object^Verb)^Predicate).
```

```
?- s(Semantics, [the, patron, ordered, a, meal], []).
```

```
Semantics = the(_4,patron(_4),a(_32,meal(_32),ordered(_4,_32)))
```



# Resolving References: exists

*A hedgehog has a nest*

```
a(X, hedgehog(X), a(Y, nest(Y), have(X, Y))).
```

```
?- hedgehog(X), a(Y, nest(Y), have(X, Y)).
```

```
exists(X, Property1, Property2) :-  
    Property1,  
    Property2,  
    !.
```



# Resolving References: all

*All hedgehogs have a nest*

```
all(X, hedgehog(X), a(Y, nest(Y), have(X, Y))).
```

*There is no hedgehog, which has no nest*

```
all(X, Property1, Property2) :-  
  \+  
  (Property1,  
  \+ Property2),  
  Property1,  
  !.
```



# Application: Spoken Language Translator (Agnäs et al. 1994)

English	<i>What is the earliest flight from Boston to Atlanta?</i>
French	<i>Quel est le premier vol Boston-Atlanta?</i>
English	<i>Show me the round trip tickets from Baltimore to Atlanta</i>
French	<i>Indiquez-moi les billets aller-retour Baltimore-Atlanta</i>
English	<i>I would like to go about nine am</i>
French	<i>Je voudrais aller aux environs de 9 heures</i>
English	<i>Show me the fares for Eastern Airlines flight one forty seven</i>
French	<i>Indiquez-moi les tarifs pour le vol Eastern Airlines cent quarante sept</i>



# Semantic Interpretation

Question:

*What is the earliest flight from Boston to Atlanta?*

Modeling a flight from Boston to Atlanta:

$\exists x(\text{flight}(x) \wedge \text{from}(x, \text{Boston}) \wedge \text{to}(x, \text{Atlanta}) \wedge \exists y(\text{time}(y) \wedge \text{departs}(x, y)))$

Finding the earliest flight:

$$\arg \min_y \exists x(\text{flight}(x) \wedge \text{from}(x, \text{Boston}) \wedge \text{to}(x, \text{Atlanta}) \wedge \exists y(\text{time}(y) \wedge \text{departs}(x, y)))$$

SLT uses the logical form as a universal representation, independent from the language.

It converts sentences from and to this representation



# Semantic Parsing

SLT does not use variables for the nouns.

*I would like to book a late flight to Boston*

is converted into the Prolog term:

```
would_like_to(i,  
  book(i,  
    np_pp(a(late(flight)),  
      X^to(X, boston))))
```



# Grammar Rules

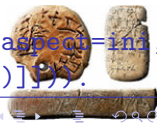
- 1 rule(s\_np\_vp,  
    s([sem=VP]),  
    [np([sem=NP,agr=Ag]),  
    vp([sem=VP,subjsem=NP,aspect=fin,agr=Ag])])).
- 2 rule(vp\_v\_np,  
    vp([sem=V,subjsem=Subj,aspect=Asp,agr=Ag]),  
    [v([sem=V,subjsem=Subj,aspect=Asp,agr=Ag,  
        subcat=[np([sem=NP])])]),  
    np([sem=NP,agr=\_])])).
- 3 rule(vp\_v\_vp,  
    vp([sem=V,subjsem=Subj,aspect=Asp,agr=Ag]),  
    [v([sem=V,subjsem=Subj,aspect=Asp,agr=Ag,  
        subcat=[vp([sem=VP,subjsem=Subj])])]),  
    vp([sem=VP,subjsem=Subj,aspect=ini,agr=\_]])).





# Lexicon

#	Lexicon entries
1	<code>lex(boston,np([sem=boston,agr=(3-s)])) .</code>
2	<code>lex(i,np([sem,agr=(1-s)])) .</code>
3	<code>lex(flight,n([sem=flight,num=s])) .</code>
4	<code>lex(late,adj([sem=late(NBAR),nbarsem=NBAR])) .</code>
5	<code>lex(a,det([sem=a(NBAR),nbarsem=NBAR,num=s])) .</code>
6	<code>lex(to,prep([sem=X^to(X,NP),npsem=NP])) .</code>
7	<code>lex(to,inf([])) .</code>
8	<code>lex(book,v([sem=have(Subj,Obj),subjsem=Subj,aspect=ini,agr=_,subcat=[np([sem=Obj])]])) .</code>
9	<code>lex(would,v([sem=would(VP),subjsem=Subj,aspect=fin,agr=_,subcat=[vp([sem=VP,aubjsem=Subj])]])) .</code>
10	<code>lex(like,v([sem=like_to(Subj,VP),subjsem=Subj,aspect=ini,agr=_,subcat=[inf([]),vp([sem=VP,subjsem=Subj])]])) .</code>



# Transferring Logical Forms

```
trule(<Comment>
  <QLF pattern 1> <Operator> <QLF pattern 2>).
```

Operator is  $\geq$ ,  $\leq$ , or  $=$ .

Lexical rules	Syntactic rules
trule([eng, fre], flight1 $\geq$ vol1).	trule([eng, fre], form(tr(relation,nn), tr(noun1), tr(noun2))  $\geq$ [and, tr(noun2), form(prepare(tr(relation)) tr(noun1))]).



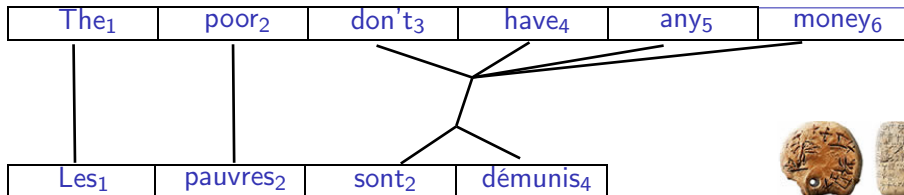
# Parallel Corpora (Swiss Federal Law)

German	French	Italian
<b>Art. 35 Milchtransport</b>	<b>Art. 35 Transport du lait</b>	<b>Art. 35 Trasporto del latte</b>
<p>1 Die Milch ist schonend und hygienisch in den Verarbeitungsbetrieb zu transportieren. Das Transportfahrzeug ist stets sauber zu halten. Zusammen mit der Milch dürfen keine Tiere und milchfremde Gegenstände transportiert werden, welche die Qualität der Milch beeinträchtigen können.</p>	<p>1 Le lait doit être transporté jusqu'à l'entreprise de transformation avec ménagement et conformément aux normes d'hygiène. Le véhicule de transport doit être toujours propre. Il ne doit transporter avec le lait aucun animal ou objet susceptible d'en altérer la qualité.</p>	<p>1 Il latte va trasportato verso l'azienda di trasformazione in modo accurato e igienico. Il veicolo adibito al trasporto va mantenuto pulito. Con il latte non possono essere trasportati animali e oggetti estranei, che potrebbero pregiudicarne la qualità.</p>



# Alignment (Brown et al. 1993)

## Canadian Hansard



# RDF and SPARQL

RDF: A popular graph format to encode knowledge.

SPARQL: A query language for RDF

In many ways, very similar to Prolog.

```
ilppp:Pierre rdf:type ilppp:person.
```

```
ilppp:Socrates rdf:type ilppp:person.
```

```
ilppp:table1 rdf:type ilppp:object.
```

```
ilppp:chair1 rdf:type ilppp:object.
```

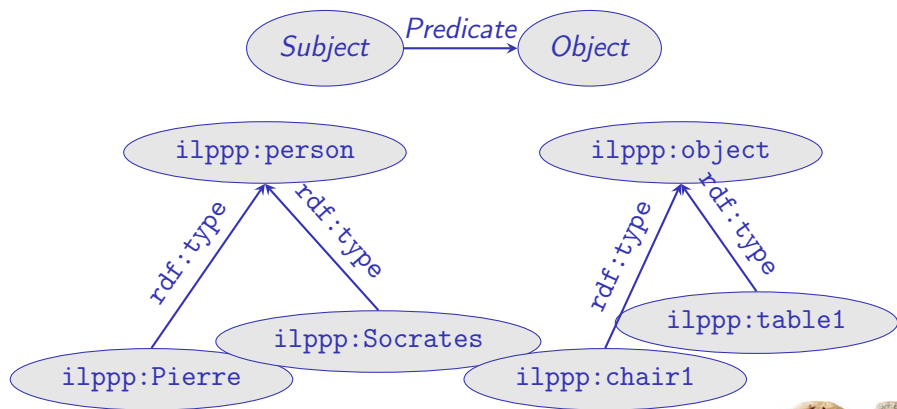
```
ilppp:chair2 rdf:type ilppp:object.
```

```
ilppp:chair1 ilppp:in_front_of ilppp:table1.
```

```
ilppp:Pierre ilppp:on ilppp:table1.
```



# RDF Triples



# RDF and SPARQL

Prolog:

```
?- object(X), object(Y), in_front_of(X, Y).
```

```
X = chair1,
```

```
Y = table1.
```

SPARQL:

```
SELECT ?x ?y
```

```
WHERE
```

```
{
```

```
  ?x rdf:type ilppp:object.
```

```
  ?y rdf:type ilppp:object.
```

```
  ?x ilppp:in_front_of ?y
```

```
}
```

Variables	?x	?y
Values	ilppp:chair1	ilppp:table1



# DBpedia, Yago, Wikidata, and Freebase

Graph databases consisting of billions of RDF triples.

Coming from a variety of sources such as Wikipedia infoboxes:

```
{{Infobox settlement
| name                = Busan
...
| area_total_km2      = 767.35
...
| population_total    = 3,614,950
...
}}
```

DBpedia: The result of a systematic triple extraction from infoboxes

```
dbpedia:Busan foaf:name "Busan, Korea"@en .
```

```
dbpedia:Busan dbpedia-owl:populationTotal "3614950".
```

```
dbpedia:Busan dbpedia-owl:areaTotal "7.6735E8" .
```

```
...
```





# SPARQL Endpoint

Network service accepting SPARQL queries such as:

```
SELECT ?entity ?population
WHERE
{
  ?entity foaf:name "Busan, Korea"@en.
  ?entity dbpedia-owl:populationTotal ?population.
}
```

that returns:

Variables	entity	population
Values	<a href="http://dbpedia.org/resource/Busan">http://dbpedia.org/resource/Busan</a>	3614950

where <http://dbpedia.org/resource/Busan> or `dbpedia:Busan` is a unique entity name based on the Wikipedia web addresses (URI nomenclature).

