

Language Technology

<http://cs.lth.se/edan20/>
Chapter 9: Phrase-Structure Grammars in Prolog

Pierre Nugues

Pierre.Nugues@cs.lth.se
http://cs.lth.se/pierre_nugues/

September 30, 2021



Constituents

The waiter brought the meal

The waiter brought the meal to the table

The waiter brought the meal of the day

Le serveur a apporté le plat

Le serveur a apporté le plat sur la table

Le serveur a apporté le plat du jour

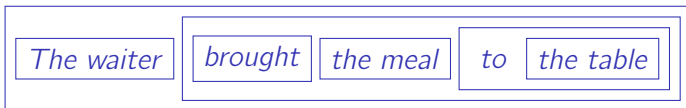
Der Ober hat die Speise gebracht

Der Ober hat die Speise zum Tisch gebracht

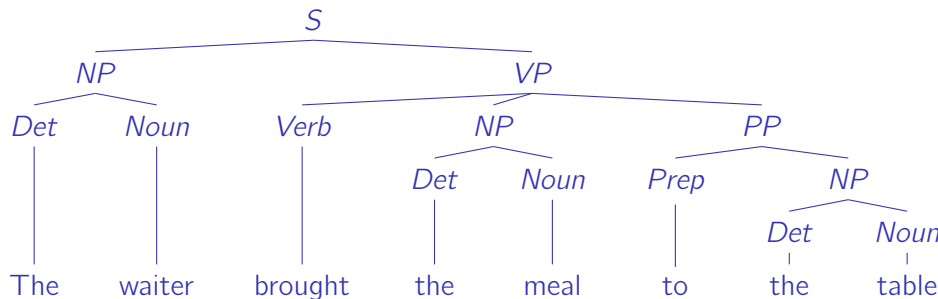
Der Ober hat die Speise des Tages gebracht



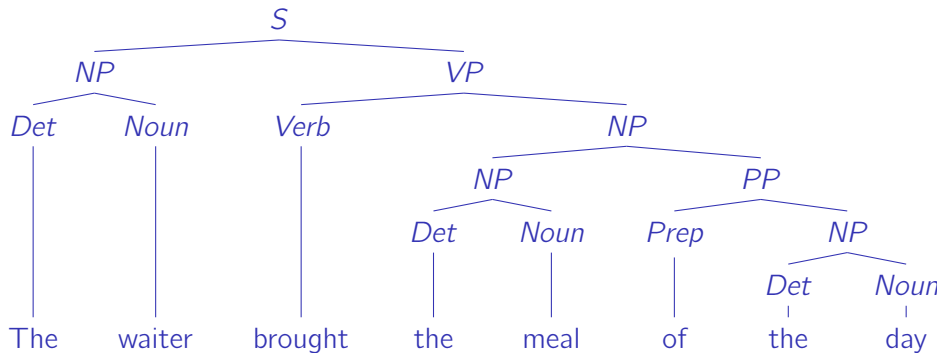
Representing Constituents



Syntactic Trees



Syntactic Trees



DCG Rules

Nonterminal symbols

`s --> np, vp, {possible_prolog_preds}.`

`np --> det, noun.`

`np --> np, pp.`

`vp --> verb, np.`

`vp --> verb, np, pp.`

`pp --> prep, np.`



DCG Rules

Terminal symbols

det --> [the].

det --> [a].

noun --> [waiter].

noun --> [meal].

noun --> [table].

noun --> [day].

verb --> [brought].

prep --> [to]. % or prep --> [to] ; [of].

prep --> [of].



Prolog Search Mechanism

Proves that a sentence is correct

```
?-s([the, waiter, brought, the, meal, to, the, table], []).  
yes.
```

```
?- s([the, waiter, brought, the, meal, of, the, day], []).  
yes.
```

Generates all the solutions

```
?-s(L, []).
```

```
L=[the, waiter, brought, the, waiter];
```

```
L=[the, waiter, brought, the, meal], etc.
```



Conversion in Prolog

`s --> np, vp.`

is translated into

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

Alternative translation:

```
s(L) :- np(L1), vp(L2), append(L1, L2, L).  
% not used
```

Terminal vocabulary:

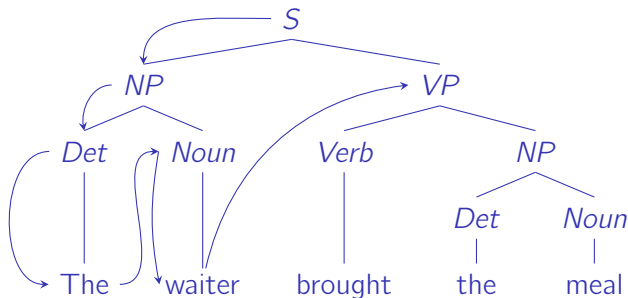
`det --> [the]`

is translated into

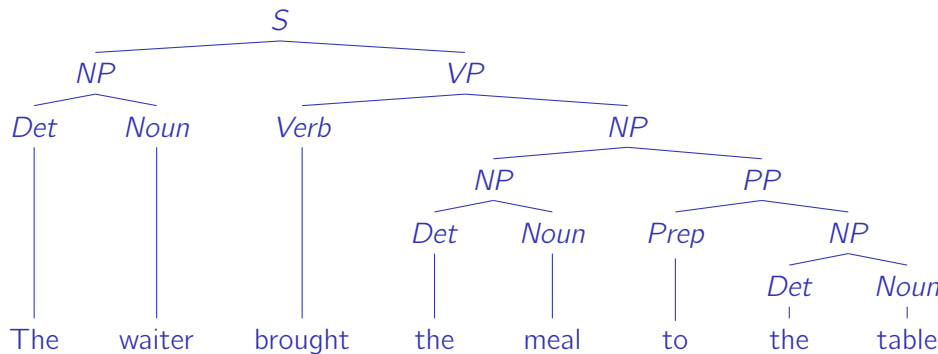
```
det(L1, L) :- c(L1, the, L).
```



The Prolog Search



Ambiguity



Left-Recursive Rules

$np \rightarrow np, pp.$

The sentence:

* *The brings the meal to the table*

traps the parser in an infinite recursion.

$npx \rightarrow det, noun.$

$np \rightarrow npx.$

$np \rightarrow npx, pp.$



Variables

```
np --> det, noun.  
det --> [le] ; [la].  
noun --> [garçon] ; [fille].
```

With variables:

```
np(Gender) --> det(Gender), noun(Gender).  
det(m) --> [le]. det(f) --> [la].  
noun(m) --> [garçon]. noun(f) --> [fille].
```



Getting the Syntactic Structure

```
s(s(NP, VP)) --> np(NP), vp(VP).  
np(np(D, N)) --> det(D), noun(N).  
vp(vp(V, NP)) --> verb(V), np(NP).
```

```
det(det(the)) --> [the].  
det(det(a)) --> [a].  
noun(noun(waiter)) --> [waiter].  
noun(noun(meal)) --> [meal].  
noun(noun(table)) --> [table].  
noun(noun(tray)) --> [tray].  
verb(verb(bring)) --> [brought].
```



Getting the Syntactic Structure

```
?-s(S, L, []).
```

Yields:

```
S = s(np(det(the), noun(waiter)),  
      vp(verb(bring), np(det(the), noun(waiter))))),  
L = [the, waiter, brought, the, waiter] ;
```



Semantic Parsing

Converts sentences to first-order logic or predicate-argument structures

Example:

Mr. Schmidt called Bill

to

`called('Mr. Schmidt', 'Bill').`

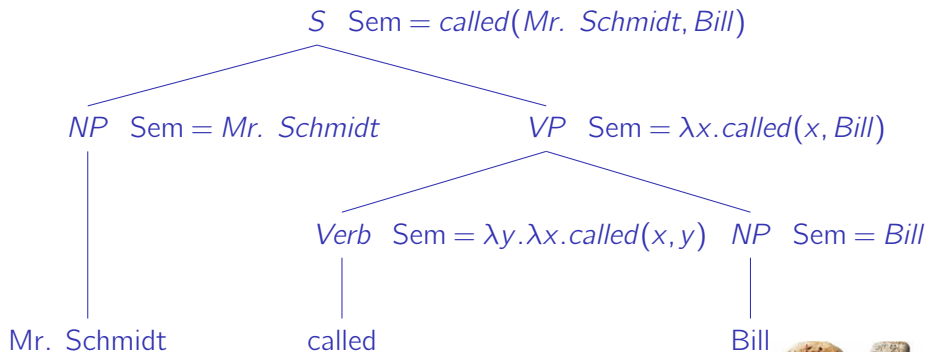
Assumption: We can compose sentence fragments (phrases) into logical forms while parsing

This corresponds to the compositionality principle



Semantic Composition

Semantic composition can be viewed as a parse tree annotation



Getting the Semantic Structure

Bill rushed `rushed('Bill').`

The verb *rushed* is represented as a lambda expression: $\lambda x.rushed(x)$

Beta reduction: $\lambda x.rushed(x)(Bill) = rushed(Bill)$

Lambda expressions are represented in Prolog as $X^rushed(X)$.

The patron ordered a meal `ordered(patron, meal)`

ordered a meal `X^ordered(X, meal)`

ordered `Y^X^ordered(X, Y)`



Getting the Semantic Structure

```
s(Semantics) --> np(Subject), vp(Subject^Semantics).  
np(X) --> det, noun(X).  
vp(Subject^Predicate) --> verb(Subject^Predicate).  
vp(Subject^Predicate) -->  
verb(Object^Subject^Predicate), np(Object).  
noun(waiter) --> [waiter].  
noun(patron) --> [patron].  
noun(meal) --> [meal]. det --> [a].  
det --> [the].
```

```
verb(X^rushed(X)) --> [rushed].  
verb(Y^X^ordered(X, Y)) --> [ordered].  
verb(Y^X^brought(X, Y)) --> [brought].
```

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



The diagram illustrates the architecture of the Persona conversational assistant. It begins with a microphone input to the 'Whisper Speech Recognition' module. This module interacts with a 'Names Database' and sends data to the 'Names Proper Name Substitution' module. The 'Names' module then feeds into the 'NLP Language Analysis' module. The 'NLP' module connects to the 'Semantic Template Matching & Object Descriptions' module, which also draws from an 'Action Template Database' and an 'Object Database (CDs)'. The 'Semantic' module's output goes to the 'Dialogue Context & Conversation State' module. This central module manages the conversation state and interacts with several other components: a 'Speech & Animation Database', the 'Player/ReActor Animation Engine', a 'Speech Controller', an 'Application CD Changer', and a 'Dialogue Rules Database'. The 'Speech Controller' is responsible for generating audio output, which is shown as a monitor and speakers. The 'Application CD Changer' is also connected to the 'Dialogue Context & Conversation State' module.



ubid=439

[//research.microsoft.com/research/pubs/view.aspx?pubid=439](https://research.microsoft.com/research/pubs/view.aspx?pubid=439)

Semantic Analysis

A command like:

Play before you accuse me by Clapton

is transformed into:

Play track1 by artist1.

The procedure is:

- 1 Produce a logical form from the parse tree:
verb(subject, object)
I would like to hear something → like(i, hear(i, something));
- 2 The logical form is transformed into the task graph:
like(i, hear(i, something)) →
verbPlay(you, objectTrack)
You play (verbPlay) a track (objectTrack)



An Example from Persona

I'd like to hear something composed by Mozart.

like1 (+Modal +Past +Futr)

Dsub: i1 (+Pers1 +Sing)

Dobj: hear1

Dsub: i1

Dobj: something1 (+Indef +Exis +Pers3 +Sing)

Prop: compose1

Dsub: mozart1 (+Sing)

Dobj: something1



Simpler Sentences

I would like something

I would like some Mozart

$s(\text{Sem}) \rightarrow np(\text{Sub}), vp(\text{Sub}^{\wedge}\text{Sem}).$

$np_x(\text{SemNP}) \rightarrow pro(\text{SemNP}).$

$np_x(\text{SemNP}) \rightarrow noun(\text{SemNP}).$

$np_x(\text{SemNP}) \rightarrow det, noun(\text{SemNP}).$

$np(\text{SemNP}) \rightarrow np_x(\text{SemNP}).$

$noun(\text{SemNP}) \rightarrow proper_noun(\text{SemNP}).$



The Verb Phrase

`verb_group(SemVG) --> verb(SemVG).`

`verb_group(SemVG) --> aux(SemAux), verb(SemVG).`

`vp(SemVP) --> verb_group(SemVP).`

`vp(SemVP) --> verb_group(Obj^SemVP), np(Obj).`

`verb(Obj^Sub^like(Sub, Obj)) --> [like].`

`verb(Obj^Sub^hear(Sub, Obj)) --> [hear].`



The Vocabulary

```
aux(would) --> [would].  
pro('I') --> ['I'].  
pro(something) --> [something].  
proper_noun('Mozart') --> ['Mozart'].  
det --> [some].  
  
?- s(Sem, ['I', would, like, some, 'Mozart'], []).  
Sem = like('I', 'Mozart')
```



More Complex Sentences

I would like to hear something

I would like to hear some Mozart

```
vp_inf(SemVP) --> [to], vp(SemVP).
```

```
vp(SemVP) --> verb_group(Obj^SemVP), vp_inf(Obj).
```

```
?- s(Sem, ['I', would, like, to, hear, some, 'Mozart'], []).
```

```
Sem = like('I', X^hear(X, 'Mozart'))
```



And Finally

```
np(SemNP) --> npx(SemVP^SemNP), vp_passive(SemVP).
```

```
vp_passive(SemVP) --> verb(Sub^SemVP) , [by], np(Sub).
```

```
verb(Sub^Obj^compose(Sub, Obj)) --> [composed].
```

```
pro(Modifier^something(Modifier)) --> [something].
```

```
?- s(Sem, ['I', would, like, to, hear, something,  
    composed, by, 'Mozart'], []).
```

```
Sem = like('I', X^hear(X, Y^something(compose('Mozart', Y))))
```

