

T-(538|725)-MALV, Natural Language Processing Context-free grammar and Prolog

Hrafn Loftsson¹ Hannes Högni Vilhjálmsón¹

¹School of Computer Science, Reykjavik University

October 2010

1 Prolog

2 Definite Clause Grammar

1 Prolog

2 Definite Clause Grammar

History

- Prolog (PROgramation en LOGique) appeared around 1970
- Built in search and unification mechanism
- Designed for natural language processing
- Since then used for various projects:
 - Database search
 - Compiler writing
 - Expert systems
 - Pattern matching

Declarative programming

- (http://en.wikipedia.org/wiki/Declarative_programming)
- Instead of using a sequence of statements (as in procedural languages)
 - Knowledge and axioms about the problem are put forth
 - The problem is modelled as a goal statement which is needed to be proved with regard to the axioms
 - Program = set of axioms
 - Calculation = proof of a goal

Material to read

- <http://en.wikipedia.org/wiki/Prolog>
- Appendix A in the textbook.
- However, we do not expect you to learn Prolog in this course.

SWI-Prolog: Download

- Go to <http://www.swi-prolog.org/>
- Download a version for your operating system
- Under Windows, user manual is:
C:\Program Files\pl\doc\windows.html

1 Prolog

2 Definite Clause Grammar

Definite Clause Grammar (DCG)

- A feature of Prolog.
- Enables us to write context-free grammar (CFG) rules directly in a Prolog program.
- Prolog converts the CFG rules to Prolog clauses.
- The prolog engine (algorithm) automatically carries out the parse without the need for additional programming.
- Main advantage of Prolog with regard to CFG: Few lines of code!

Definite Clause Grammar (DCG)

An example

Rules	Lexicon
s --> np, vp.	det --> [the]. verb --> [brought].
np --> det, noun.	det --> [a]. prep --> [to].
np --> np, pp.	noun --> [waiter]. prep --> [of].
vp --> verb, np.	noun --> [meal].
vp --> verb, np, pp.	noun --> [table].
pp --> prep, np.	noun --> [day].

Usually, the operator `:-` is used as a separator between the left-hand side and the right-hand side of a Prolog clause. In the case of DCG, `-->` is used.

Definite Clause Grammar (DCG)

Prolog search

- The Prolog search mechanism checks whether a fact is true or generates all the solutions.
- In the case of DCG, Prolog checks if a given sentence can be derived from the grammar or generates all the sentences accepted by the grammar.

Test

- Start Prolog: `"c:\Program Files\pl\bin\plwin"`
- Load af file (program): `?- consult('cfg.pro').`

Definite Clause Grammar (DCG)

Is a given string in the language?

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

Yes

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

Yes

Generate all (grammatically correct) sentences

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

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

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

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

```
...
```

Translating DCGs into Prolog clauses

- The DCG rule `s --> np, vp`.
- ... is translated to the clause:
- `s(L) :- np(L1), vp(L2)`.
-
- Example: *the waiter brought the meal*
 - L matches [the, waiter, brought the meal]
 - L1 matches the noun phrase [the, waiter]
 - L2 matches the verb phrase [brought, the, meal]

Translating DCGs into Prolog clauses

To be precise this is implemented with so-called *difference lists*

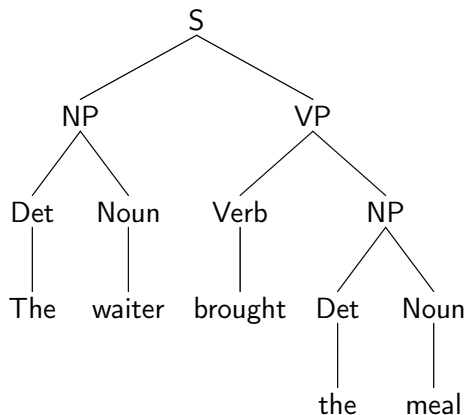
- The DCG: `s --> np, vp.`
- ... is translated to the clause:
- `s(L1,L) :- np(L1,L2), vp(L2,L).`
- The lexical DCG rule: `det --> [the].`
- ... is translated into the fact:
- `det([the | L], L).`

Constituent values correspond to the difference of each pair of arguments

- $s(L1,L) :- np(L1,L2), vp(L2,L).$
- *The waiter brought the meal* corresponds to the s symbol and unifies with $L1 \setminus L$, where $L1 \setminus L$, denotes $L1$ minus L .
- *The waiter* corresponds to the np symbol and unifies with $L1 \setminus L2$.
- *brought the meal* corresponds to the vp symbol and unifies with $L2 \setminus L$.
- In terms of lists
 - $L1 \setminus L$ corresponds to [the, waiter brought, the meal]
 - $L1 \setminus L2$ corresponds to [the, waiter]
 - $L2 \setminus L$ corresponds to [brought, the, meal]

Prolog performs *depth-first search*

- ?- trace.
- ?- s([the, waiter, brought, the, meal], []).



Prolog performs *depth-first search*

- `?- s([the, waiter, brought, the, meal], []).`
- `s(L1, L) :- np(L1, L2), vp(L2, L).`
 - `L1 = [the, waiter, brought, the, meal], L=[]`
- `np(L1, L) :- det(L1, L2), noun(L2, L).`
- `det([the | L], L).`
 - `L = [waiter, brought, the, meal].`
 - $\Rightarrow L1 = \text{the}$
 - $\Rightarrow L2 = [\text{waiter, brought, the, meal}]$
- `noun([waiter, brought, the, meal],L).`
- ...

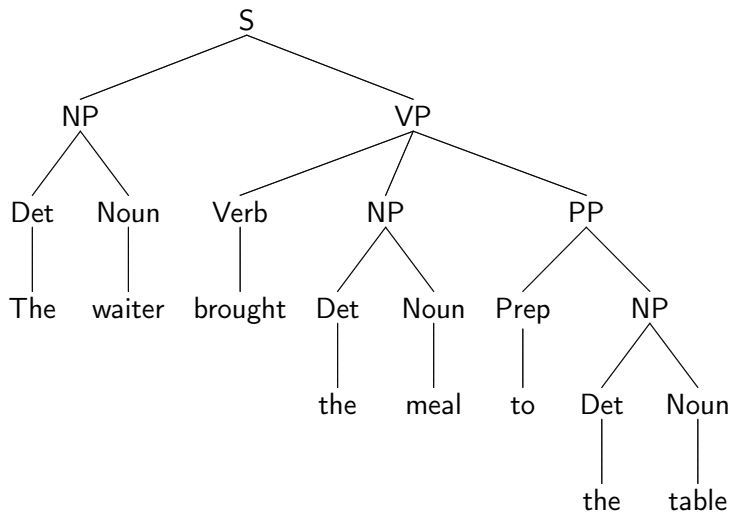
Left recursion (í. vinstri endurkvæmni)

- ?- s([the, brought, the, meal], []).
ERROR: Out of local stack
- Due to the left-recursive rule: $np \rightarrow np, pp$.
- Left-recursiveness can be eliminated from a grammar (see Section 8.3.3 in the textbook).

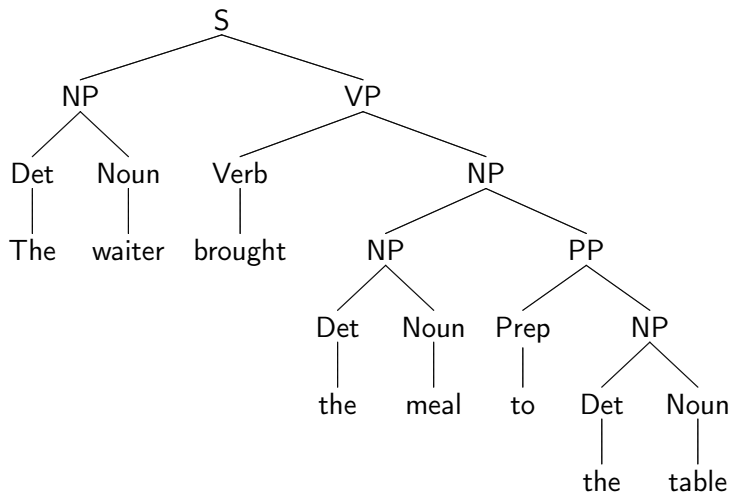
Parsing ambiguity (í. margræðni í þáttun)

- The rules ...
- $vp \rightarrow verb, np$ and $np \rightarrow np, pp$
- $vp \rightarrow verb, np, pp$
- ... lead to ambiguity, because more than one parse tree is possible for a given string.
- For example the string: *The waiter brought the meal to the table*
- Tree 1a and 2a in the following slides are correct.

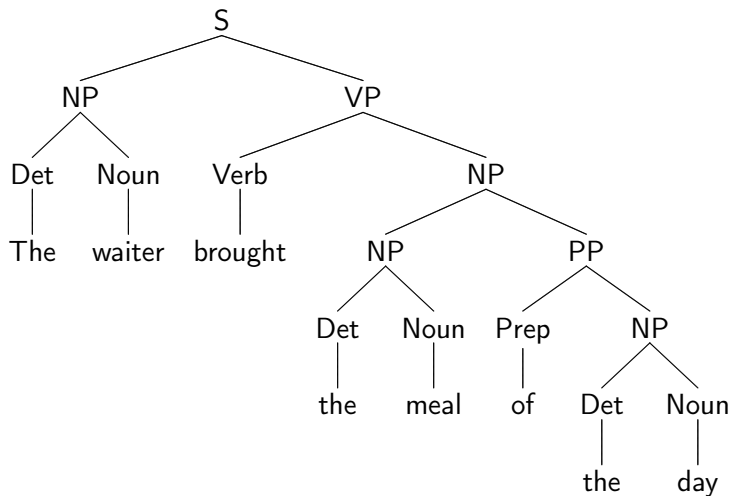
Parsing ambiguity – Tree 1a



Parsing ambiguity – Tree 1b



Parsing ambiguity – Tree 2a



Parsing ambiguity – Tree 2b

