Definite Clause Grammars

Begoña Villada and Gosse Bouma

May 13, 2003

Today's lecture

- Grammars, sentences and trees
- Parsing as deduction
- Parsing in Prolog
- Word order
- Definite Clause Grammars (DCG)
- (Number and gender) agreement
- Beyond Context-Free Grammars

Context-Free Grammar

Grammar rules	Lexicon
	NP $\longrightarrow$ dit
S $\longrightarrow$ NP VP	Det $\longrightarrow$ een
NP $\longrightarrow$ Det Adj N	Adj $\longrightarrow$ eenvoudig
VP $\longrightarrow$ V	N $\longrightarrow$ voorbeeld
VP $\longrightarrow$ V NP	V $\longrightarrow$ is

Linking grammar, sentences and trees

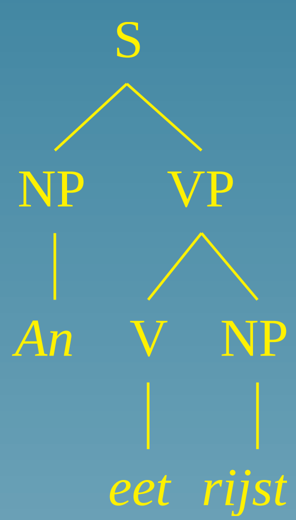
What is the relationship between grammars, and the languages they generate or the constituent structure they represent?

S → NP VP
An eet rijst

Linking grammar, sentences and trees

What is the relationship between grammars, and the languages they generate or the constituent structure they represent?

S → NP VP
An eet rijst



A CFG grammar rule

can be interpreted

- as a string rewriting condition
  - ⋆ a sequence xyz can be re-written as xRHSz iff y = **LHS** and LHS $\longrightarrow$ RHS is a rule in the CFG grammar
  - ⋆ the language of a CFG is the set of sequences of terminal symbols that can be rewritten from the distinguished start symbol S
- or as a tree admissability condition
  - ⋆ a tree is admitted by a CFG iff for each **local domain** in the tree there is a grammar rule from which the local tree can be *projected*
  - ⋆ a local domain corresponds to a node in the tree and its immediate daughters

Parsing as deduction

Both interpretations of a grammar rule

- tree licensing condition
- (string re-writing or) language generating condition

have much in common with deduction:

- given known facts, other information can be concluded on the basis of the grammar rules

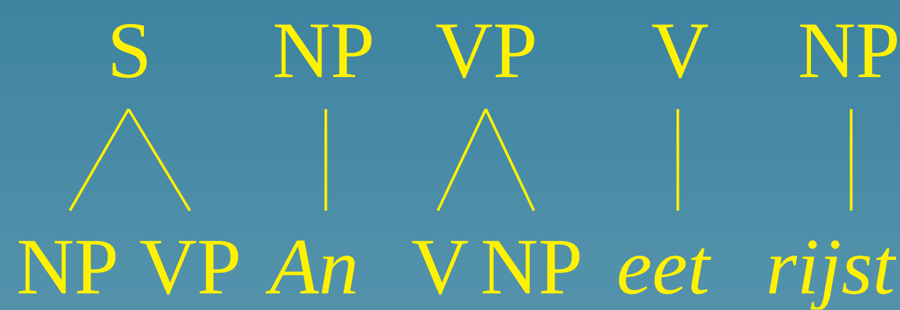
Given a string

- An eet rijst met kip ('An eats rice with chicken')

Is the string in the language recognized by the following CFG?

Grammar rules	Lexicon
	NP → An
S → NP VP	N → kip
NP → N PP	P → met
NP → N	N → rijst
PP → P NP	V → eet
VP → V NP	V → eten

Tree licensing or string re-writing



Parsing in Prolog: from grammar rules to clauses

A re-write CFG grammar rule translates into a Prolog rule

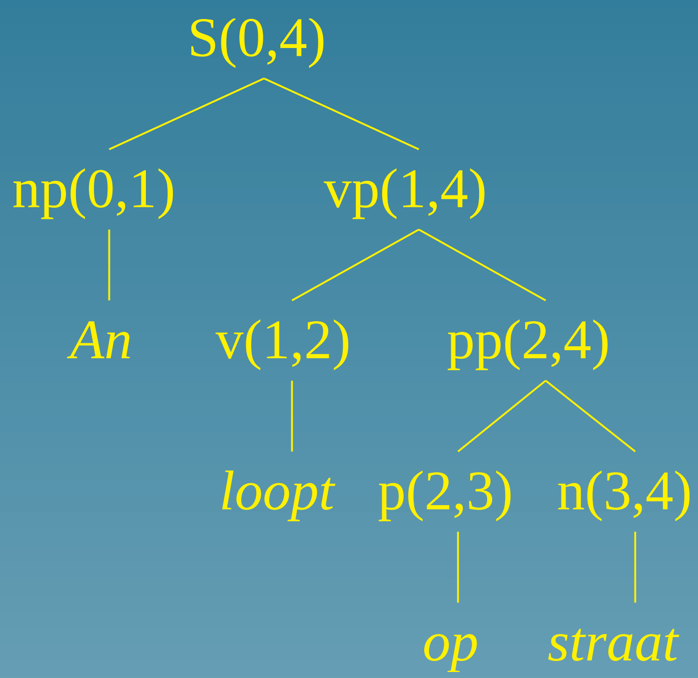
- S $\longrightarrow$ NP VP
- `s :- np, vp.`

Read as: ‘we can build an S if we have built an NP and a VP’

Grammar rules vs. Prolog clauses

- difference between re-write rule and Prolog: (word) order
- the following Prolog rules are 'equivalent':
- ```
  s :- np, vp.
  s :- vp, np.
  s :- np, vp, np.
  ```
- the corresponding re-write rules are not.
  ```

Word order



Adding word order

- s(P0,P3) : there is an s between position P0 and P3.
- `s(P0,P3) :- np(P0,P1), vp(P1,P3).`
- Rules that introduce words:
  - ⋆
    ```
    np(P0, P1) :-  woord(P0, an,    P1).
    v(P1,  P2) :-  woord(P1, eet,   P2).
    np(P2, P3) :-  woord(P2, rijst, P3).
    ```
- give (assert) input like:
  - ⋆ `woord(0,an,1).  woord(1,eet,2).  woord(2,rijst,3).`

String positions as lists

Instead of representing positions using a pair of integers, we use a pair of lists whose difference represents the segment of text in question:

- Input: [an,eet,rijst]
- Position 0: [an,eet,rijst]
- Position 1: [eet,rijst]
- Position 2: [rijst]
- Position 3: [] (also: end of the sentence)

Positions as lists

Hence the initial query takes the form

- ```
  ? s([an,eet,rijst],  []).
  ```

and the next queries:

- ```
      ? np([an,eet,rijst], P1).

          ? woord([an,eet,rijst], an, P1).
  ```

The general rule:

- ```
  woord([Woord|Wn], Woord , Wn).
  ```
  ```

Definite clause grammar

- each re-write rule consists of two arguments that encode string positions
- rules that introduce a word remove (pop) the first word from the list
- these two mechanisms are built-in in the Prolog DCG-notation

Definite clause grammar II

- `s --> np, vp.`

  translates as

- `s(P0, P2) :- np(P0, P1), vp(P1, P2).`

- `np --> [an].`

  translates as

- `np(P0, P1) :- 'C'(P0, an, P1).`

  ('C' is Sicstus' predicate for woord).

Example DCG

Query:

? s([het, kind, koopt, een, ijsje, in, het, park], []).

Grammar rules	Lexicon
	det $\longrightarrow$ [een].
	det $\longrightarrow$ [het].
s $\longrightarrow$ np, vp.	n $\longrightarrow$ [kind].
np $\longrightarrow$ det, n.	n $\longrightarrow$ [ijsje].
vp $\longrightarrow$ v, np.	n $\longrightarrow$ [park].
vp $\longrightarrow$ v, np, pp.	v $\longrightarrow$ [koopt].
vp $\longrightarrow$ v, pp.	v $\longrightarrow$ [loopt].
pp $\longrightarrow$ p, np.	p $\longrightarrow$ [in].

Strings as difference lists (1)

Difference lists give us an alternative way to write CFG-rules in Prolog:

```
s(String) :-
          append(NP, VP, String),
          np(NP),
          vp(VP).

np([jan]).

vp(String):-
          append(V,NP,String),
          v(V),
          np(NP).
```

Advantage: non-deterministic use of append !

Strings as difference lists (2)

You can also see the string positions of a DCG as a difference-lists, that make append **unnecessary**.

```
s(In, Out) :-
            np(In, Mid),
            vp(Mid, Out).

np([jan|Rest],Rest).

vp(In, Out):-
            v(In, Mid),
            np(Mid, Out).
```

Using Features: agreement

Our DCG should allow:	but disallow:
an loopt op straat	*an loop op straat
an walks along the street	*an walk along the street
wij gaan fietsen	*wij gaat fietsen
we cycle	*we cycles

Agreement in DCG

Basic idea add arguments to handle the agreement features: P(erson) and N(umber).

- `s --> np(P,N), vp(P,N).`
- `vp(P,N) --> v(P,N), pp.`

Where does the information about person and number come from?

Agreement lexically specified

- Lexical entry:

```
np(3,sg) --> [an].
np(3,pl) --> [wij].
v(3,sg) --> [loopt].
v(3,pl) --> [gaan].
```

- ```
  s(P0, P2) :- np(P, N, P0, P1), vp(P, N, P1, P2).
  ```
  ```

De/Het agreement

```
np     --> de(Det), n(Det).

det(de)  --> [de].

det(het) --> [het].

det(_)   --> [een].

n(de)    --> [hond].

n(het)   --> [hondje].
```

A/an agreement in English

```
np  --> de(Det), n(Det).

det(a)  --> [a].

det(an) --> [an].

det(_)  --> [the].

n(a)    --> [cat].

n(an)   --> [hour].
```

Verbs select their arguments

Verbs set restrictions on the arguments they select

- ⋆ Wim slaapt / *Wim slaapt Ben
  ⋆ Wim kent Ben / *Wim kent
  ⋆ Wim denkt aan Ben / *Wim denkt van Ben

- ⋆ Wim sleeps / *Wim sleeps Ben
  ⋆ Wim knows Ben / *Wim knows
  ⋆ Wim put the book on the table / *Wim put the book

Argument selection

Grammar rules:

```
vp --> v(intrans).
vp --> v(trans), np.
vp --> v(Prep), pp(Prep).
pp(Prep) --> p(Prep), np.
```

Lexicon:

```
v(intrans) --> [slaapt].
v(trans) --> [kent].
v(aan) --> [denk].
p(aan) --> [aan].
```

Beyond CFG

- CFG with features (attributes) and unification
- Examples: Definite Clause Grammar and Unification Grammar
- Transformational Grammar is problematic for computational purposes:
  - ⋆ little formal precision
  - ⋆ problematic for automatic analysis

Beyond CFG (contd.)

- de language WW (a string of words followed by the same string of words) is not context-free:
  - ⋆ aabcccaabccc
- Dutch verbal clusters show comparable cross-serial dependencies:
  - ⋆ dat Peter Hans Cecilia de kraanvogels zag helpen fotograferen
- Dutch (and Swiss-German) admit subordinate clauses in which all the verbs follow all the nouns

Curly brackets

Sometimes we combine pure Prolog-code and DCG-notation:

```
v(1,sg) --> [Word], {ww(Word,_,_)}.
v(2,sg) --> [Word], {ww(_,Word,_)}.
v(3,sg) --> [Word], {ww(_,_,Word)}.
```

```
ww(ben, bent, is).
ww(heb, hebt, heeft).
```

```
v(1, sg, P0, P1) :- 'C'(P0,Word,P1), ww(Word,_,_).
```