Definite Clause Grammar (DCG)

Übersicht

- Formale Sprachen
- Kontextfreie Grammatiken (*Context-Free Grammars*)
  - Terminale, Nichtterminale und Regeln
- Kontextfreie Grammatik in Form von DCGs
- Formales Ableiten von Satzformen und Sätzen
- phrase/2: DCG-Standardparser in Prolog
- Prologinterne Repräsentation und Verabeitung von DCGs
- Parsing-Strategie: Top-Down und Left-Right
- Problem: Linksrekursive Grammatiken
  - Linguistische Motivation für linksrekursive Grammatiken
  - Abhilfen

Formale Sprachen

Vokabular *T* einer Sprache (Terminale)

- $T_{Englisch}$ = {aardvark,…,cat,…,woman,…,zymurgy}

*T** ist die Menge aller endlichen Folgen des Vokabulars *T*.

- $T_{Englisch}^*$ =
  { ε,
  a,aardvark,cat,woman,… *Folge aus 0 Elementen (epsilon)*
  a cat, cat a, peter sleeps,… *Folgen aus 1 Element*
  a a a, a cat sleeps, woman a cat,… *Folgen aus 2 Elementen*
  …} *Folgen aus 3 Elementen*
  Folgen aus n Elementen

Sprache *L* über Vokabular *T* ist Teilmenge von *T**

- $L_{Englisch}$ = {yes,…,peter sings,…,a cat sleeps,…,the man loves her,…}

▶ Mit Grammatiken kann die gewünschte Teilmenge von T* formal und elegant spezifiziert werden.

Kontextfreie Grammatiken (CFG)

Beispiele für Regeln einer kontextfreien Grammatik:

S → NP VP	*Det → the*	*V → loves*
VP → V NP	*Det → a*	*V → sings*
VP → V	*N → cat*	*V → sees*
NP → Det N	*N → woman*	*V → thinks*

Gemäss diesen Grammatikregeln sind etwa folgende Sätze (S) erlaubt bzw. nicht erlaubt (*)

- a cat sings
- *woman sees cat
- the woman loves a cat

Kontextfreie Regeln

Bestandteile einer kontextfreien Regel:

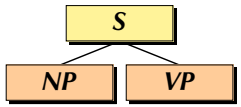
S → *NP VP*

linke Seite
left-hand side

rechte Seite
right-hand side

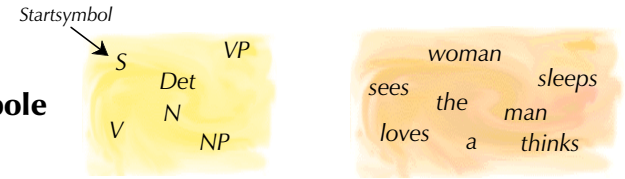
Lesarten:

- Ein S besteht aus einer NP und einer VP.
- Eine NP gefolgt von einer VP ergibt ein S.
- S expandiert zu NP und VP.



CFG: Symbolisch-Graphisch…

Symbole



Nicht-Terminal-Symbole
"Syntaktische Kategorien"

Terminal-Symbole
"Lexikon"

Regeln

S → *NP VP*	*Det* → *the*	*V* → *loves*
NP → *Det N*	*Det* → *a*	*V* → *sings*
VP → *V NP*	*N* → *cat*	*V* → *sees*
VP → *V*	*N* → *woman*	*V* → *thinks*

CFG: Mathematisch…

Jede **kontextfreie Grammatik** lässt sich durch ein 4-Tupel (*N, T, R, S*) beschreiben, wobei gilt:

- *N* ist endliche Menge von **Nicht-Terminal-Symbolen**
- *T* ist endliche Menge von **Terminalsymbolen** ($T \cap N = \emptyset$)
- *R* ist endliche Menge von **Regeln** ($R \subseteq N \times (N \cup T)^*$)
- *S* ist das **Startsymbol** ($S \in N$)

Pfeilnotation $A \rightarrow \alpha$ für Regeln

- ist lesbarere Schreibvariante für Tupel (A, α), mit $A \in N$ und $\alpha \in (N \cup T)^*$

CFG im DCG-Formalismus

Regeln mit Nicht-Terminalen auf der rechten Seite:

- Syntaktische Regeln

```
s --> np, vp.
np --> det, n.
```

```
vp --> v.
vp --> v, np.
```

Regeln mit Terminal-Symbolen auf der rechten Seite:

- Lexikalische Regeln

```
det --> [the].
det --> [a].
```

```
n --> [cat].
v --> [sees].
v --> [sings].
```

- Verwendung von Terminalen und Nicht-Terminalen in der RHS wäre möglich, macht die Sache aber unübersichtlicher.

Ableiten von Sätzen

Satzformen einer Grammatik *G*

- *S* ist eine **Satzform**, falls *S* das Startsymbol von *G* ist.
- *S* ist eine **Satzform** der Form $\alpha\delta\gamma$, falls gilt
  - es gibt eine Satzform der Form $\alpha B\gamma$, und
  - es gibt eine Regel der Form $B \rightarrow \delta$.

S ⇒ NP VP ⇒ Det N VP ⇒ Det N V ⇒ the N V ⇒ the cat V ⇒ the cat sings

Sätze einer Grammatik *G*

- *S* ist ein **Satz** von *G*, falls gilt:
  - *S* ist eine Satzform von *G*, und
  - *S* enthält nur Terminalsymbole von *G*.

DCG-Parsen mit phrase/2

Das eingebaute Prädikat phrase/2 überprüft, ob von einem Nicht-Terminal eine Liste von Terminalsymbolen abgeleitet werden kann.

```
?- phrase(s, [the,cat,sings]).
yes
```

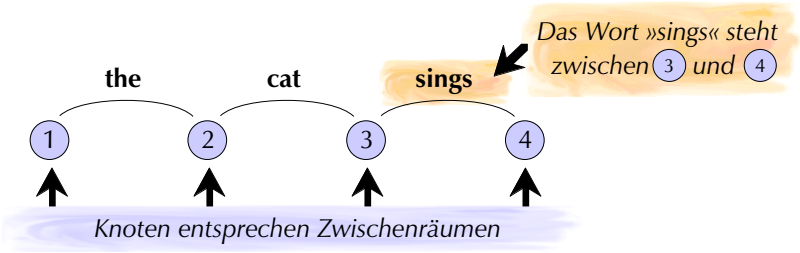
```
?- phrase(np, [a,cat]).
yes
```

```
?- phrase(s, [a,cat]).
no
```

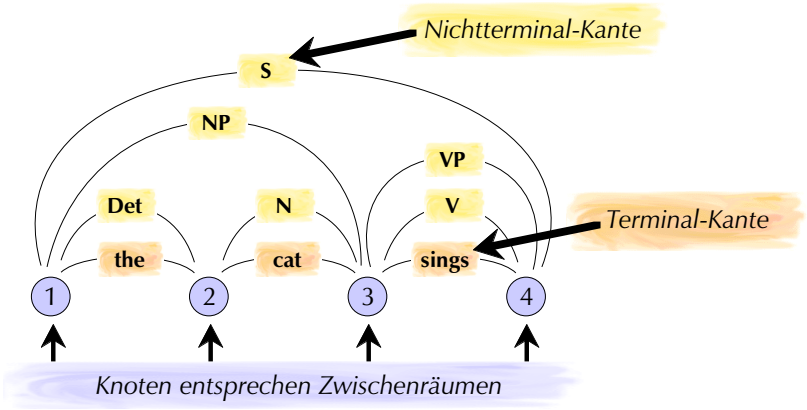

Graphische Veranschaulichung

Anhand einer Grafik lässt sich die Grundidee zeigen, wie der in Prolog eingebaute Parser für DCGs funktioniert

- Die Start-/Zwischen-/Endpositionen werden vergegenständlicht!
- Prologs Parser speichert seine Daten jedoch *nicht* numerisch!



Graphische Veranschaulichung



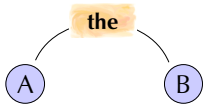
Einlesezauber I: DCG zu Prolog

Beim Einlesen (»Konsultieren«) eines Programms werden DCG-Regeln in gewöhnliche Prolog-Klauseln übersetzt.

- Regel mit Terminal-Symbolen auf der rechten Seite:

```
det --> [the].
```

```
det(A, B) :-
  'C'(A, the, B).
```



Die Knoten A und B sind durch »the« verbunden.

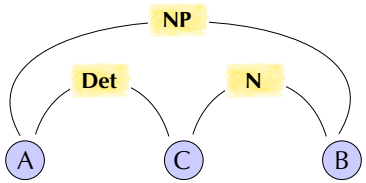
Einlesezauber II: DCG zu Prolog

Beim Einlesen (»Konsultieren«) eines Programms werden DCG-Regeln in gewöhnliche Prolog-Klauseln übersetzt.

- Regel mit Nichtterminalen auf der rechten Seite:

```
np --> det, n.
```

```
np(A, B) :-
  det(A, C),
  n(C, B).
```

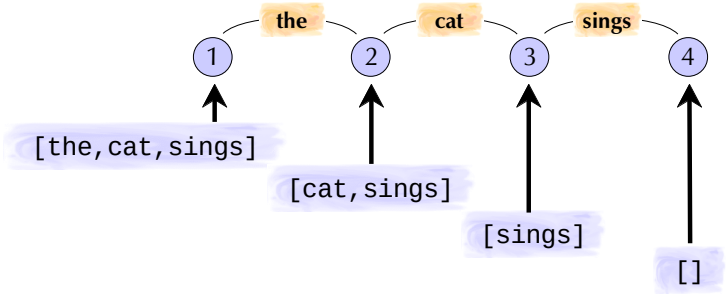


Damit zwischen A und B eine NP steht, braucht es zwischen A und einem Knoten C ein Det, sowie zwischen C und B ein N.

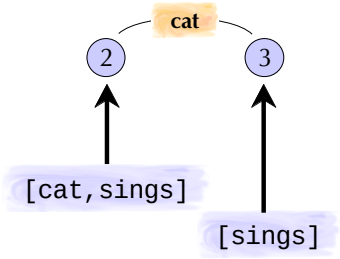
Repräsentation der Zwischenknoten

Für Knoten stehen Listen mit dem Rest des Satzes.

- Jeder Knoten ist damit eindeutig identifiziert.



'C'/3: C heisst *connect*



In Prolog vordefiniert

```
'C'([Word|Rest], Word, Rest).
```

Der allgemeine Fall: Wann verbindet ein Wort zwei Knoten?

```
'C'([cat,sings], cat, [sings]).
```

»cat« verbindet Knoten 2 und 3.

Einlesezauber durch Termexpansion

Beim Einlesen (»Konsultieren«) eines Programms werden DCG-Regeln in gewöhnliche Prolog-Klauseln übersetzt.

```
s --> np, vp.
np --> det, n.
vp --> v.
vp --> v, np.
```

```
s(A,B)  :- np(A,C), vp(C,B).
np(A,B) :- det(A,C), n(C,B).
vp(A,B) :- v(A,B).
vp(A,B) :- v(A,C), np(C,B).
```

```
det --> [the].
det --> [a].
n --> [cat].
v --> [sees].
v --> [sings].
```

```
det(A,B) :- 'C'(A, the, B).
det(A,B) :- 'C'(A, a, B).
n(A,B)   :- 'C'(A, cat, B).
v(A,B)   :- 'C'(A, sees, B).
v(A,B)   :- 'C'(A, sings, B).
```

DCG-Atome als Prädikate

An das Ergebnis dieser Übersetzung können Prolog-Anfragen gestellt werden:

```
?- det([the,cat], [cat]).
yes
```

Terminale

```
det(A,B) :- 'C'(A, the, B).
det(A,B) :- 'C'(A, a, B).
n(A,B)   :- 'C'(A, cat, B).
v(A,B)   :- 'C'(A, sees, B).
v(A,B)   :- 'C'(A, sings, B).
```

```
?- np([the,cat], []).
yes

?- s([the,cat,sings], []).
yes
```

Nichtterm.

```
s(A,B)  :- np(A,C), vp(C,B).
np(A,B) :- det(A,C), n(C,B).
vp(A,B) :- v(A,B).
vp(A,B) :- v(A,C), np(C,B).
```

vordef.

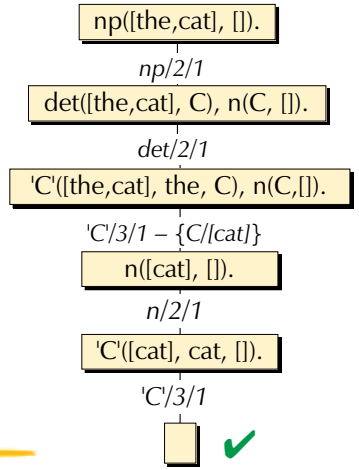
```
'C'([Word|Rest], Word, Rest).
```


DCG: Parsen heisst Beweisen

Schritte zum Beweis von

```
?- np([the,cat], []).
yes
```

Wegen der *Top-Down*- und *Left-Right*-Strategie des Prolog Beweisers arbeitet der DCG-Parser die zu analysierende Kette ebenfalls *Top-Down* und *Left-Right* ab.



Linksrekursive Grammatiken

```
np --> np, conj, np.

conj --> [and].
```

```
np(A, B) :-
   np(A, C),
   conj(C, D),
   np(D, B).

conj(A, B) :-
   'C'(A, and, B).
```

Was geschieht bei folgender Anfrage?

```
?- phrase(np,[a,cat,and,a,cat]).
   1  1  Call: phrase(user:np,[a,cat,and,a,cat]) ?
   2  2  Call: np([a,cat,and,a,cat],[]) ?
   3  3  Call: np([a,cat,and,a,cat],_1210) ?
   4  4  Call: np([a,cat,and,a,cat],_1616) ?
...
```


Linksrekursive Grammatiken

Wo liegt das Problem?

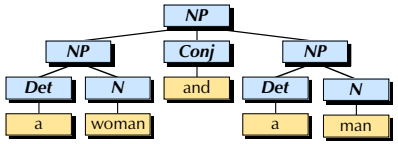
- der Parser gerät in einen endlose Schleife
- weil von einem Nicht-Terminal eine Kette abgeleitet werden kann, die wiederum mit demselben Nicht-Terminal beginnt
- der Parser springt von einem Nicht-Terminal zum nächsten, ohne ein Terminalsymbol zu konsumieren

Derartige Grammatiken heissen *links-rekursiv*.

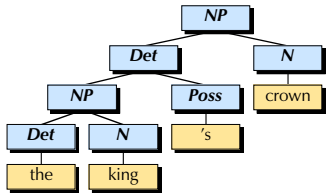
- Links-Rekursion ist für Top-Down-Parser, die von links nach rechts arbeiten, ein Problem.

Linksrekursive Grammatiken

LinguistInnen brauchen linksrekursive Grammatiken.



$$NP \rightarrow NP \;\; Conj \;\; NP$$



$$NP \rightarrow Det \;\; N$$
$$Det \rightarrow NP \;\; Poss$$

Linksrekursive Grammatiken

Mögliche Abhilfen

- linksrekursive Grammatiken verbieten
- Grammatik so umwandeln, dass sie nicht mehr linksrekursiv ist
  - dies ist für jede kontextfreie Grammatik möglich
  - aber die den Sätzen zugewiesene Struktur ist dann nicht mehr so, wie sich das die LinguistInnen wünschen
- ein anderes Parsing-Verfahren verwenden, das mit links-rekursiven Grammatiken zurecht kommt

In der Praxis wird häufig die dritte Variante gewählt

- denn andere Parsing-Verfahren sind effizienter als reine Top-Down-Algorithmen (hängt sehr stark von Implementation ab)