

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 Verarbeitung von DCGs
- ◆ Parsing-Strategie: Top-Down und Left-Right
- ◆ Problem: Linksrekursive Grammatiken
 - ◆ Linguistische Motivation für linksrekursive Grammatiken
 - ◆ Abhilfen

DCG - 1

Formale Sprachen

Vokabular T einer Sprache (Terminale)

◆ $T_{\text{Englisch}} = \{\text{aardvark}, \dots, \text{cat}, \dots, \text{woman}, \dots, \text{zymurgy}\}$

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

◆ $T_{\text{Englisch}}^* =$
 $\{ \epsilon,$
 $a, \text{aardvark}, \text{cat}, \text{woman}, \dots$
 $a \text{ cat}, \text{cat } a, \text{peter sleeps}, \dots$
 $a \ a \ a, \text{a cat sleeps}, \text{woman a cat}, \dots$
 $\dots \}$

Folge aus 0 Elementen (epsilon)
 Folgen aus 1 Element
 Folgen aus 2 Elementen
 Folgen aus 3 Elementen
 Folgen aus n Elementen

Sprache L über Vokabular T ist Teilmenge von T^*

◆ $L_{\text{Englisch}} = \{\text{yes}, \dots, \text{peter sings}, \dots, \text{a cat sleeps}, \dots, \text{the man loves her}, \dots\}$

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

DCG - 2

Kontextfreie Grammatiken (CFG)

Beispiele für Regeln einer kontextfreien Grammatik:

$S \rightarrow NP VP$	$Det \rightarrow the$	$V \rightarrow loves$
$VP \rightarrow V NP$	$Det \rightarrow a$	$V \rightarrow sings$
$VP \rightarrow V$	$N \rightarrow cat$	$V \rightarrow sees$
$NP \rightarrow Det N$	$N \rightarrow woman$	$V \rightarrow thinks$

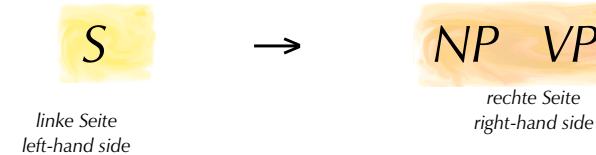
Gemäss diesen Grammatikregeln sind etwa folgende Sätze (S) erlaubt bzw. nicht erlaubt (*)

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

DCG - 3

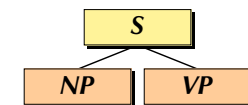
Kontextfreie Regeln

Bestandteile einer kontextfreien Regel:



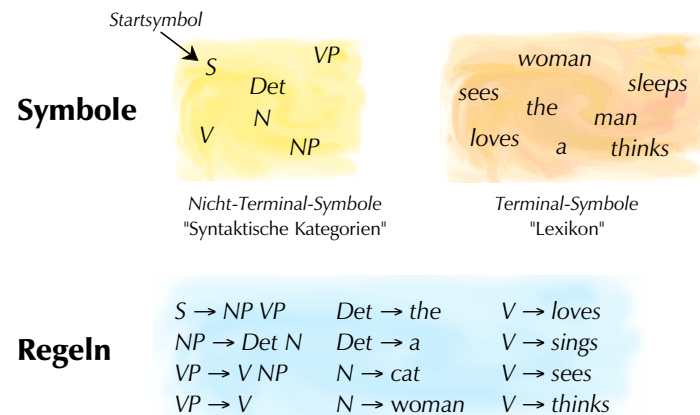
Lesarten:

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



DCG - 4

CFG: Symbolisch-Graphisch...



DCG - 5

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 $\alpha \in (N \cup T)^*$

DCG - 6

CFG im DCG-Formalismus

Regeln mit Nicht-Terminalen auf der rechten Seite:

- ♦ Syntaktische Regeln

$s \rightarrow np, vp.$
 $np \rightarrow det, n.$
 $vp \rightarrow v.$
 $vp \rightarrow v, np.$

Regeln mit Terminal-Symbolen auf der rechten Seite:

- ♦ Lexikalische Regeln

$det \rightarrow [the].$
 $det \rightarrow [a].$
 $n \rightarrow [cat].$
 $v \rightarrow [sees].$
 $v \rightarrow [sings].$

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

DCG - 7

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 $B \rightarrow \delta$.

$S \Rightarrow NP VP \Rightarrow Det N VP \Rightarrow Det N V \Rightarrow the N V \Rightarrow the\ cat V \Rightarrow the\ cat\ sings$

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 - 8

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
```

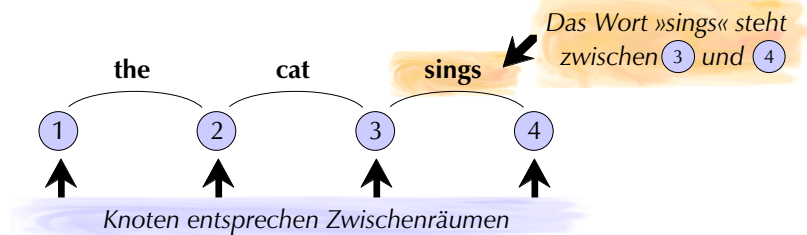
```
?- phrase(s, [a, cat]).
no
```

DCG-9

Graphische Veranschaulichung

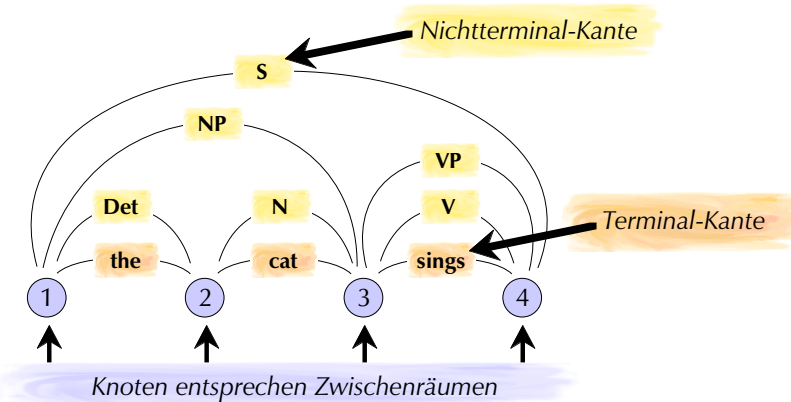
Anhand einer Grafik lässt sich die Grundidee zeigen, wie der in Prolog eingebaute Parser für DCGs funktioniert

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



DCG-10

Graphische Veranschaulichung



DCG-11

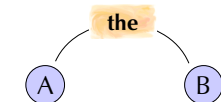
Einlesezauber I: DCG zu Prolog

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).
```



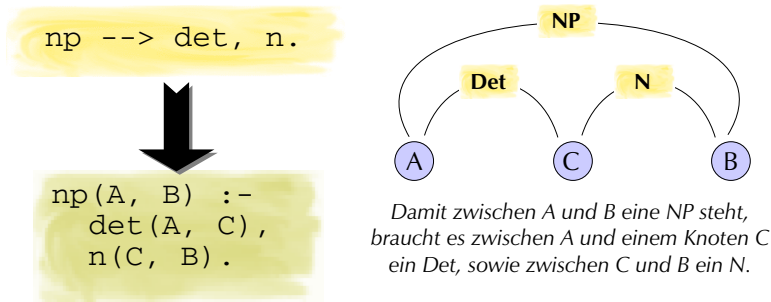
Die Knoten A und B sind durch »the« verbunden.

DCG-12

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:

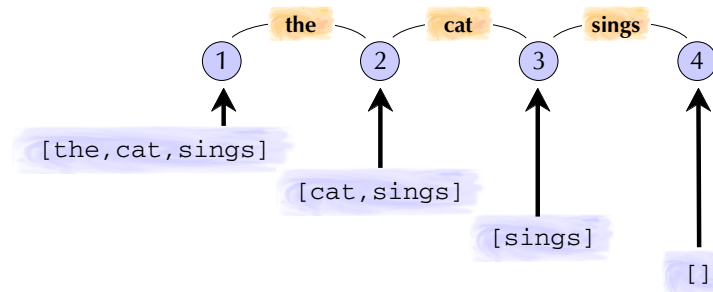


DCG-13

Repräsentation der Zwischenknoten

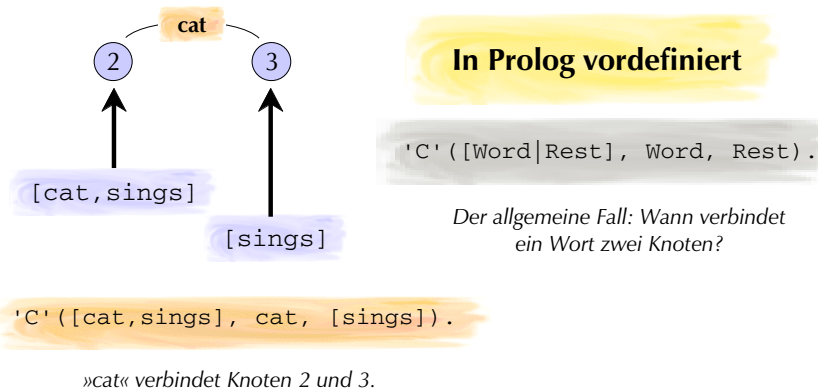
Für Knoten stehen Listen mit dem Rest des Satzes.

- ♦ Jeder Knoten ist damit eindeutig identifiziert.



DCG-14

'C'/3: C heisst connect



DCG-15

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.`

\longrightarrow

`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].`

\longrightarrow

`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-16

DCG-Atome als Prädikate

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

<pre>?- det([the,cat], [cat]). yes</pre>	Terminale <pre>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).</pre>
<pre>?- np([the,cat], []). yes</pre>	Nichtterm. <pre>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).</pre>
<pre>?- s([the,cat,sings], []). yes</pre>	vordef. <pre>'C'([Word Rest], Word, Rest).</pre>

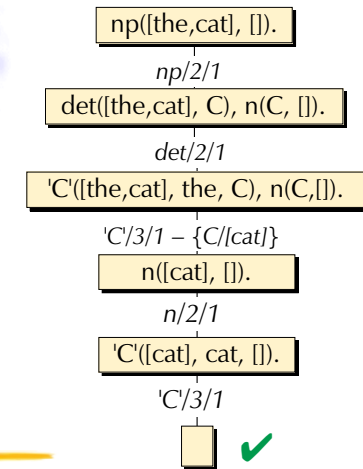
DCG-17

DCG: Parsen heisst Beweisen

Schritte zum Beweis von

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

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



DCG-18

Linksrekursive Grammatiken

<pre>np --> np, conj, np.</pre>		<pre>np(A, B) :- np(A, C), conj(C, D), np(D, B).</pre>
<pre>conj --> [and].</pre>		<pre>conj(A, B) :- 'C'(A, and, B).</pre>

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) ?
...
```

DCG-19

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

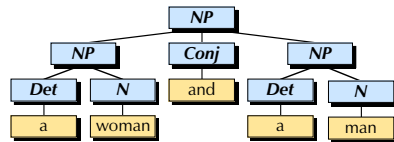
Derartige Grammatiken heissen *links-rekursiv*.

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

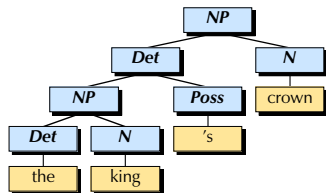
DCG-20

Linksrekursive Grammatiken

LinguistInnen brauchen linksrekursive Grammatiken.



$NP \rightarrow NP \text{ Conj } NP$



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

DCG - 21

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)

DCG - 22