Top-Down-Parsing: DCGs

Übersicht

- Formalismus
- Benutzung des in Prolog eingebauten Top-Down-Parsers für DCGs
- Funktionsweise
- Komplexe Nichtterminalsymbole
  - Modellieren von Kongruenz
  - Ausgabe der erkannten Struktur
  - Auch nicht-kontextfreie Grammatiken können mit DCGs modelliert werden
- Einbetten von Prolog-Klauseln
- Problem: Linksrekursive Grammatiken
  - Linguistische Motivation für linksrekursive Grammatiken
  - Abhilfen

Definit-Klausel-Grammatiken (DCGs)

In Prolog ist ein einfacher Top-Down-Parser bereits eingebaut.

- Grammatik besteht aus Prolog-Klauseln (Definiten Klauseln)
- nützlich zum schnellen Ausprobieren einer Mini-Grammatik

Allerdings:

- eher ineffizientes Verfahren
  - bei mehrdeutigen Grammatiken werden unter Umständen Teile des Satzes mehrmals analysiert
- »Aufhängen« bei bestimmten Grammatiken
- ⇒ für richtige Sprachverarbeitungsprojekte so gut wie unbrauchbar

DCGs: Formalismus

Regeln mit Nichtterminalen auf der rechten Seite:

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

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

Regeln mit Terminal-Symbolen auf der rechten Seite:

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

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

DCGs: Benutzung

Ein eingebautes Prädikat phrase/2 überprüft, ob von einem Nichtterminal eine Kette von Terminalsymbolen abgeleitet werden kann.

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

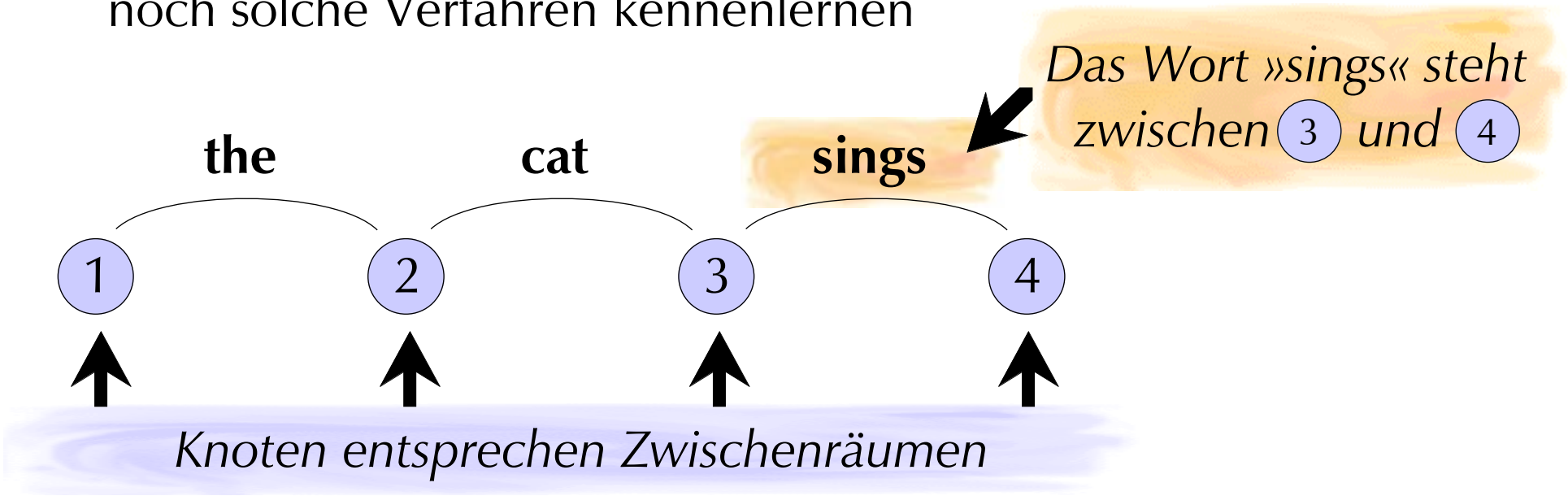
```
?- phrase(np, [a, cat]).
yes
```

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

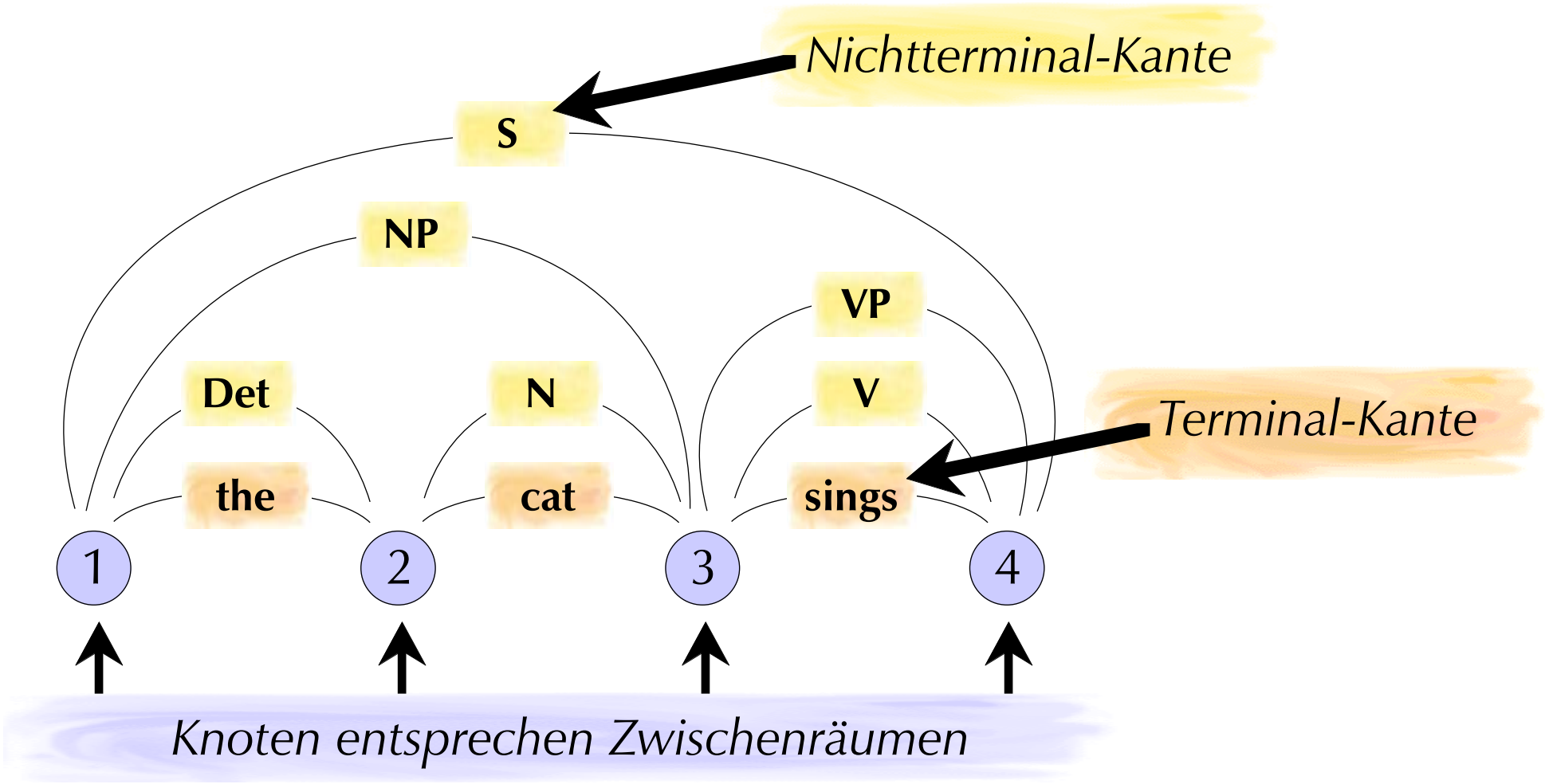
Graphische Veranschaulichung

Anhand einer Grafik lässt sich erklären, wie der in Prolog eingebaute Parser für DCGs funktioniert

- Prologs Parser speichert seine Daten jedoch *nicht* in dieser Weise
- andere Parsing-Verfahren hingegen schon; wir werden im Sommer noch solche Verfahren kennenlernen



Graphische Veranschaulichung

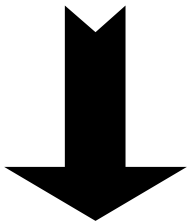


DCGs: Funktionsweise

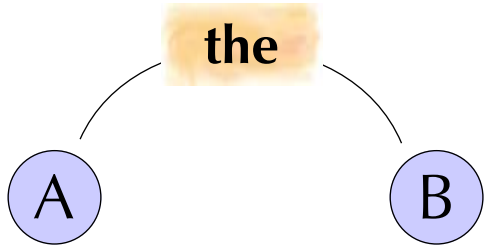
Beim Einlesen (»Konsultieren«) eines Programms werden DCG-Regeln in gewöhnliche Prolog-Klauseln übersetzt.

- Regel mit Terminal-Symbolen auf der rechten Seite:

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



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



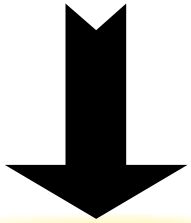
Die Knoten A und B sind durch »the« verbunden.

DCGs: Funktionsweise

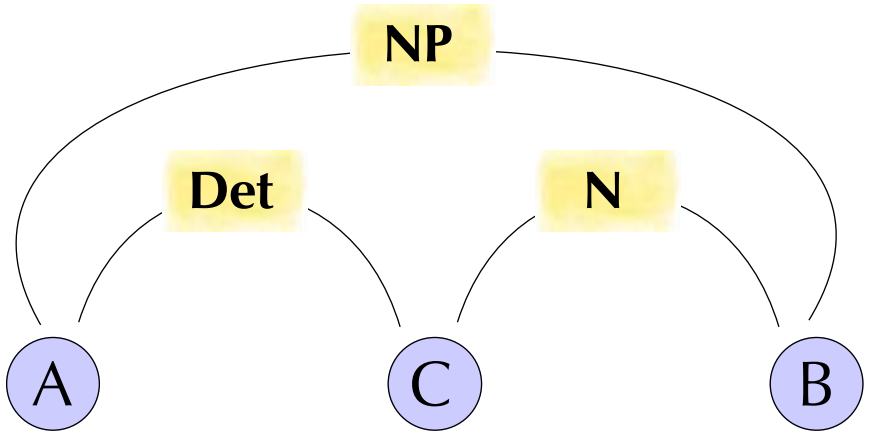
Beim Einlesen (»Konsultieren«) eines Programms werden DCG-Regeln in gewöhnliche Prolog-Klauseln übersetzt.

- Regel mit Nichtterminalen auf der rechten Seite:

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



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

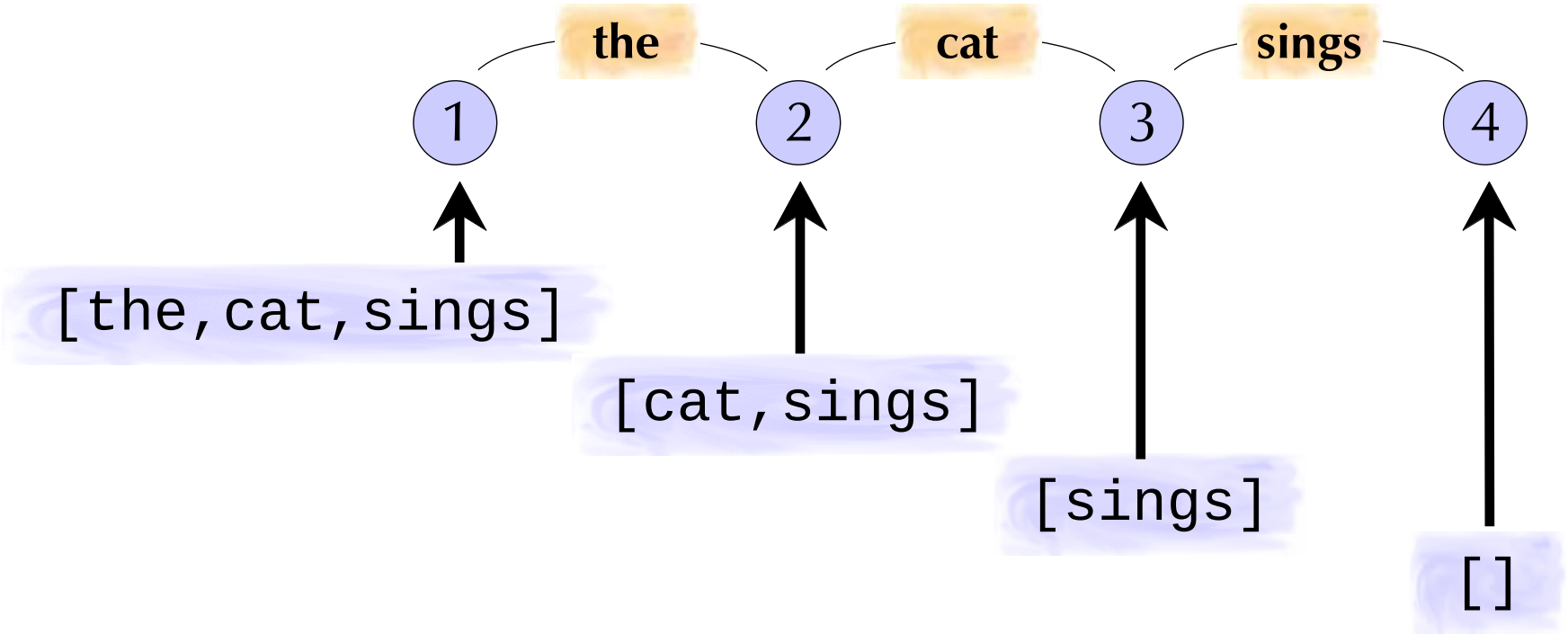


Damit zwischen A und B eine NP steht, braucht es zwischen A und einem Knoten C ein Det, sowie zwischen C und B ein N.

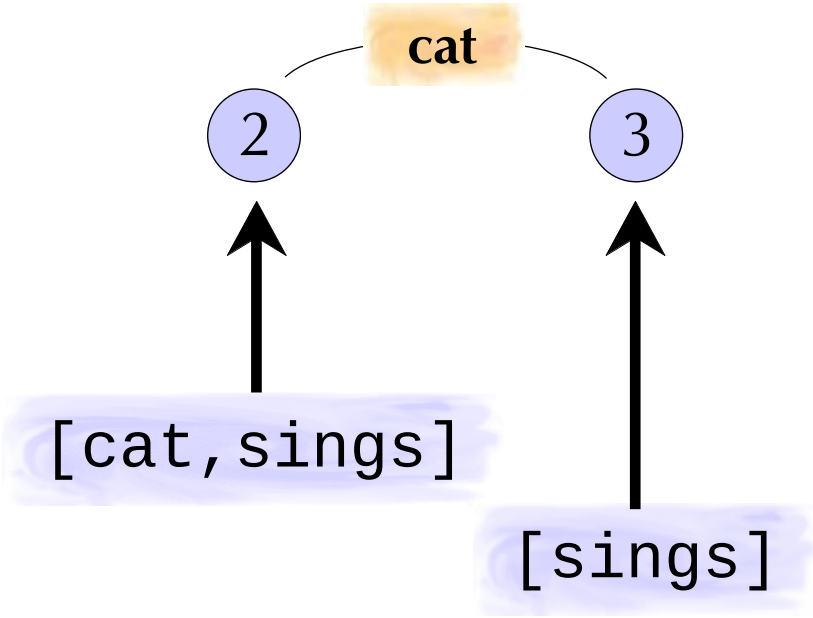
DCGs: Funktionsweise

Für Knoten stehen Listen mit dem Rest des Satzes.

- »Differenzlisten«: effiziente Prolog-Programmiertechnik



DCGs: Funktionsweise




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

»cat« verbindet Knoten 2 und 3.

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

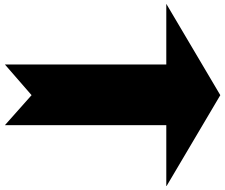
Der allgemeine Fall: Wann verbindet ein Wort zwei Knoten?

In Prolog vordefiniert

DCGs: Funktionsweise

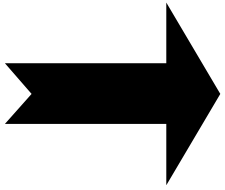
Beim Einlesen (»Konsultieren«) eines Programms werden DCG-Regeln in gewöhnliche Prolog-Klauseln übersetzt.

```
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).
```

DCGs: Funktionsweise

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

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

```
?- 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).
```

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).
```

vordef.

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

DCGs: Funktionsweise

Schritte zum Beweis von np([the,cat], []):

- `np([the,cat], [])`
- `det([the,cat], X), n(X, [])`
- `'C'([the,cat], the, X), n(X, [])`
- `n([cat], [])`
- `'C'([cat], cat, [])`
-

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).
```

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).
```

vordef.

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

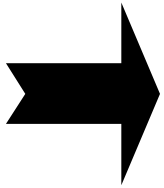
Komplexe Nichtterminal-Symbole

Bisher waren alle Symbole in der DCG Prolog-Atome

- Der DCG-Formalismus kennt jedoch auch komplexe Nichtterminale
- Argumente können unifiziert werden
  - Modellieren von Kongruenz-Phänomenen *(Agreement)*

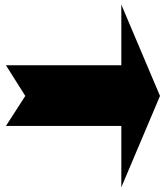
    *dem Katze — Genus von Artikel und Nomen muss übereinstimmen

```
np(Genus) -->
  det(Genus),
  n(Genus).
```



```
np(Genus, X, Y) :-
  det(Genus, X, Z),
  n(Genus, Z, Y).
```

```
det(m) --> [der].
```



```
det(m, X, Y) :-
  'C'(X, der, Y).
```

Komplexe Nichtterminal-Symbole

- Ausgabe eines Syntaxbaums

Bisher: nur »yes« oder »no« (d.h. ob Satz grammatisch ist oder ungrammatisch)

Grammatik

```
s(satz(StrukturDerNP, StrukturDerVP)) -->
  np(StrukturDerNP), vp(StrukturDerVP).
np(nominalphrase(StrukturDet, StrukturN)) -->
  det(StrukturDet), n(StrukturN).
vp(verbalphrase(StrukturV)) --> v(StrukturV).
vp(verbalphrase(StrukturV, StrukturNP)) -->
  v(StrukturV), np(StrukturNP).
det(artikel(the)) --> [the].
n(nomen(cat)) --> [cat].
v(verb(sees)) --> [sees].
v(verb(sings)) --> [sings].
```

Anfrage

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

Baum = satz(nominalphrase(artikel(the),
                          nomen(cat)),
            verbalphrase(verb(sings)))
```

Komplexe Nichtterminal-Symbole

Komplexe Nichtterminal-Symbole beeinflussen die mathematischen Eigenschaften des Formalismus

- die modellierte Sprache kann ausserhalb der Klasse der kontextfreien Sprachen liegen

Grammatik für $\{a^n\, b^n\, c^n \mid n \geq 1\}$

```
s --> as(N), bs(N), cs(N).

as(1) --> [a].
as(x(E)) --> [a], as(E).

bs(1) --> [b].
bs(x(F)) --> [b], bs(F).

cs(1) --> [c].
cs(x(G)) --> [c], cs(G).
```

Anfragen

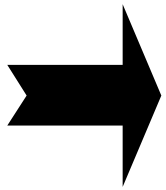
```
?- phrase(s, [a,b,c]).
yes

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

Eingebettete Prolog-Klauseln

Terme in geschweiften Klammern

- werden bei der Übersetzung von DCGs nach Prolog-Prädikaten unverändert übernommen
- dadurch können Prolog-Programme in die Grammatik eingebettet werden

```
zahl --> [N],
  {number(N)}.
```



```
zahl(X, Y) :-
  'C'(X, N, Y),
  number(N).
```

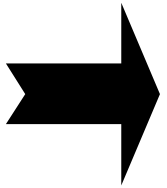
Eingebettete Prolog-Klauseln

Jedes beliebige Prolog-Prädikat kann in eine DCG eingebettet werden.

- dadurch können auch nicht-kontextfreie Sprachen erkannt werden
- keine deklarative Spezifikation der Grammatik
  - Grammatiken werden schnell unübersichtlich
  - eine DCG mit eingebettetem Prolog muss von Prolog verarbeitet werden; ein Parser für normale DCGs könnte hingegen problemlos auch in einer anderen Programmiersprache geschrieben sein
    - genaugenommen könnte man natürlich eine DCG mit eingebettetem Prolog durchaus mit anderen Programmiersprachen verarbeiten
    - aber man müsste dann eben die gesamte Prolog-Umgebung simulieren

Linksrekursive Grammatiken

```
a --> b.
b --> a, c.
c --> [bla].
```



```
a(X, Y) :- b(X, Y).
b(X, Y) :- a(X, Z), c(Z, Y).
c(X, Y) :- 'C'(X, bla, Y).
```

Was geschieht bei der Anfrage phrase(a, [bla,bla])?

- a([bla,bla], [ ])
- b([bla,bla], [ ])
- a([bla,bla], …)
- b([bla,bla], …)
- etc.

Linksrekursive Grammatiken

Wo liegt das Problem?

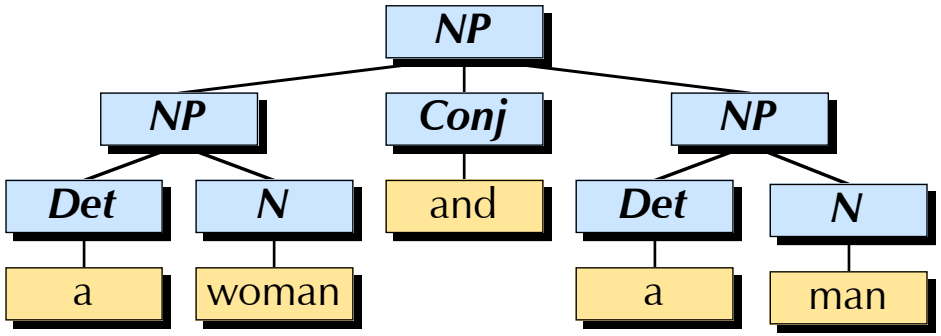
- der Parser gerät in eine endlose Schleife
- weil von einem Nicht-Terminal eine Kette abgeleitet werden kann, die wiederum mit demselben Nicht-Terminal beginnt
  - a ⇒ b ⇒ a c ⇒ b c ⇒ a c c ⇒ b c c ⇒ a c c c ⇒ b c c c ⇒ a c c c c
- der Parser springt somit von einem Nichtterminal zum nächsten, ohne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

Derartige Grammatiken heissen *links-rekursiv*.

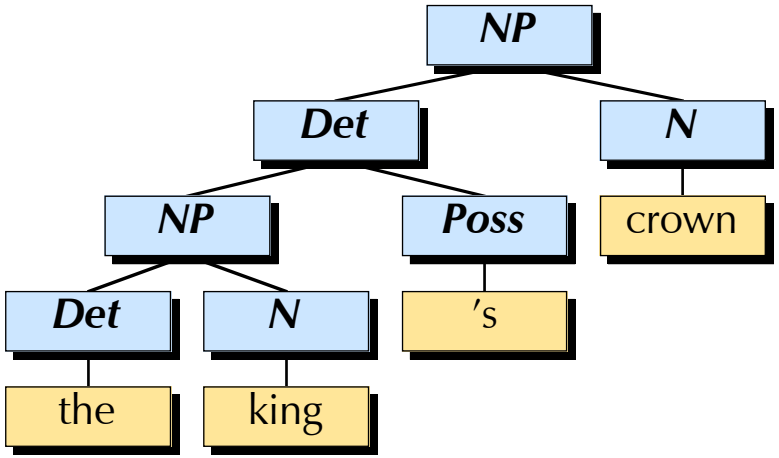
- Links-Rekursion ist für gewöhnliche Top-Down-Parser ein Problem.

Linksrekursive Grammatiken

Linguisten wollen linksrekursive Grammatiken schreiben.



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



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

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 Linguisten wünschen
- ein Parsing-Verfahren verwenden, das mit links-rekursiven Grammatiken zurechtkommt (d.h. keinen Top-Down-Parser benutzen)

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)

Aufgaben: DCGs

Programmiertechniken der Computerlinguistik 1 · Wintersemester 1998/99

1. Singende Katzen

Angenommen, die in der Vorlesung benutzte Katzen-DCG wäre bereits konsultiert worden: Welche einzelnen Schritte nimmt Prolog vor, um die Anfrage

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

zu beweisen? Zeichne den entsprechenden Beweisbaum.

2. Tilgungsregeln

In einer DCG können auch Grammatikregel formuliert werden, deren rechte Seite leer ist. Derartige Regeln werden Tilgungs- oder ε-Regeln genannt. Im DCG-Formalismus können sie wie folgt formuliert werden (natürlich kann statt »leer« auch ein anderer Term stehen):

```
leer --> [].
```

Lies eine Grammatik mit einer Tilgungsregeln ein und überprüfe durch Aufruf von listing/0, zu welcher Prolog-Klausel diese Regel übersetzt wird. Stellen Tilgungsregeln ein Problem für den in Prolog eingebauten Top-Down-Parser dar?

3. Reden ist Silber

Konsultiere die Katzen-DCG, stelle die folgende Anfrage und löse durch Eingabe des Strichpunkts Backtracking aus. Was geschieht?

```
?- phrase(s, Satz).
```

4. Komplexe Nichtterminale

In der ersten Aufgabe des Aufgabenblattes »Parsing-Einführung« ging es darum, eine kontextfreie Grammatik für einfache deutsche Sätze zu erstellen.

a) Formuliere Deine Grammatik in Form einer DCG und probiere sie aus, indem Du einige Sätze durch den in Prolog eingebauten Top-Down-Parser analysieren lässt.

b) Modelliere Kongruenz-Phänomene, indem Du komplexe Nichtterminale einführst und einzelne Argumente unifizierst. So soll sichergestellt sein, dass Artikel und Nomen in einer NP denselben Kasus, Numerus und Genus besitzen, um fehlerhafte NPs wie »dem Katze« auszuschliessen.

c) Unterscheide zwischen transitiven und intransitiven Verben. Bei ersteren muss ein Akkusativobjekt im Satz vorhanden sein, bei letzteren darf gar kein Objekt stehen. So sind »Die Katze sieht die Maus« und »Die Katze schläft« grammatische Sätze, nicht aber »Die Katze sieht« und »Die Katze schläft die Maus«.

d) Erweitere Deine Grammatik so, dass die Struktur der erkannten Sätze zurückgegeben wird.

5. Linksrekursion

Angenommen, Du bist aus irgendeinem Grund dazu verpflichtet, einen Top-Down-Parser zu verwenden, und die Grammatik sei linksrekursiv und dürfe aus linguistischen Gründen nicht verändert werden.

Gibt es trotzdem eine Möglichkeit, einen Top-Down-Parser zu schreiben, der bei jedem Eingabesatz terminiert?