GOETHE UNIVERSITÄT FRANKFURT AM MAIN

www.uni-frankfurt.de

Einführung in die Methoden der Künstlichen Intelligenz

Logisches Programmieren

PD Dr. David Sabel

SoSe 2014

Stand der Folien: 29. Mai 2023

Resolution als Programmausführung

- Prädikatenlogische Relation kann zum Programmieren verwendet werden
- Relation als Abbildung
- $P(x, y)$: x als Eingabe, y als Ausgabe, aber auch umgekehrt.
- Inetwa $f(x) = x + 1$ als $P(x, x + 1)$
- Zunächst keine Funktion (da nicht eineindeutig)
- Beachte $x + 1$ wird als Term interpretiert, nicht als Zahl

Beispiel

Wissenbasis bzw. Definitionen / Fakten:

$$\{vater(peter, maria)\},$$
$$\{mutter(susanne, maria)\},$$
$$\{vater(peter, monika)\},$$
$$\{mutter(susanne, monika)\},$$
$$\{vater(karl, peter)\},$$
$$\{mutter(elisabeth, peter)\},$$
$$\{vater(karl, pia)\},$$
$$\{mutter(elisabeth, pia)\},$$
$$\{vater(karl, paul)\},$$
$$\{mutter(elisabeth, paul)\}$$

„Programmiere“ Eltern:

$$\forall X, Y : vater(X, Y) \implies eltern(X, Y)$$
$$\wedge$$
$$\forall X, Y : mutter(X, Y) \implies eltern(X, Y)$$

als CNF

$\{eltern(X, Y), \neg vater(X, Y)\}$ und $\{eltern(X, Y), \neg mutter(X, Y)\}$

Beispiel (2)

Anfrage / Aufruf: $\exists Y.eltern(karl, Y)$ (hat Karl Kinder?)

Beispiel (2)

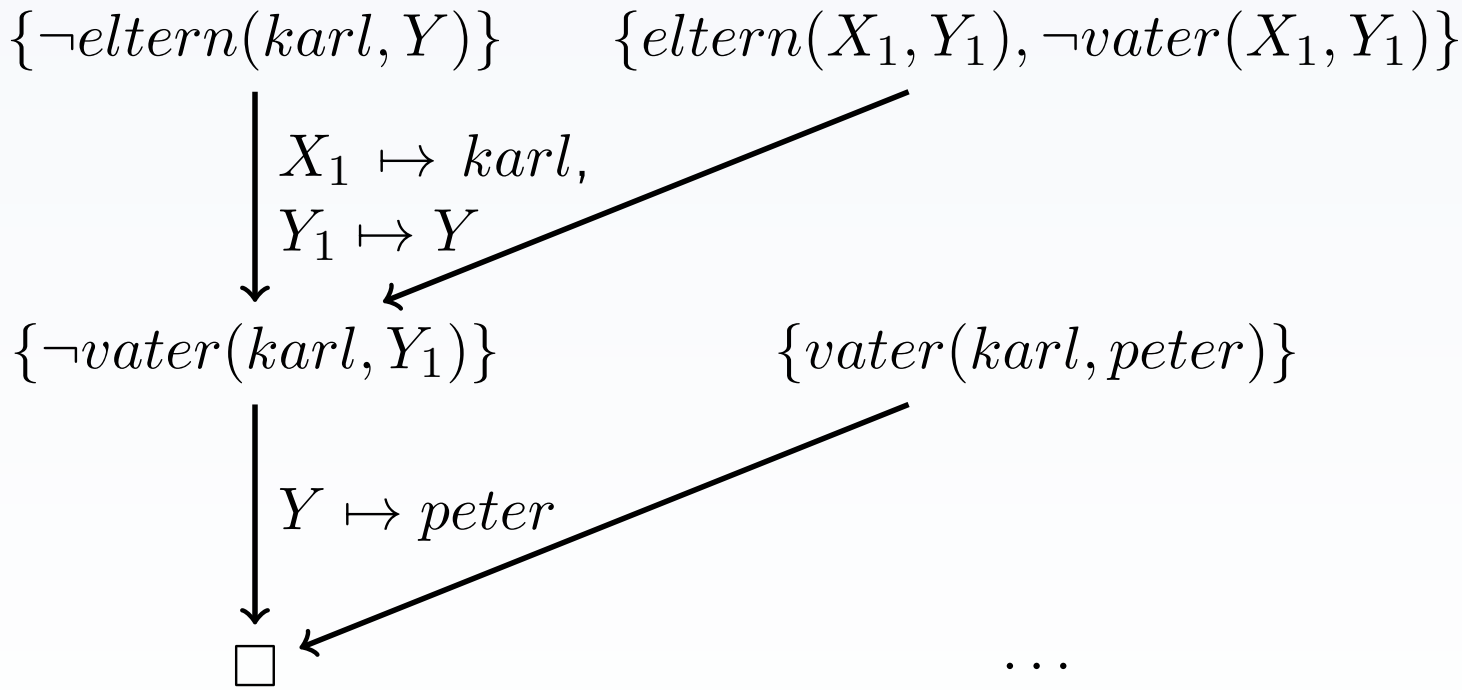
Anfrage / Aufruf: $\exists Y.eltern(karl, Y)$ (hat Karl Kinder?)

- Zeige: $\mathcal{F} \models G$ mit Resolution: $\mathcal{F}, \neg G$ ist widersprüchlich
- $\neg\exists Y.eltern(karl, Y)$ wird zur Anfrageklausel $\{\neg eltern(karl, Y)\}$
- Verwende lineare Resolution

Beispiel (2)

Anfrage / Aufruf: $\exists Y.eltern(karl, Y)$ (hat Karl Kinder?)

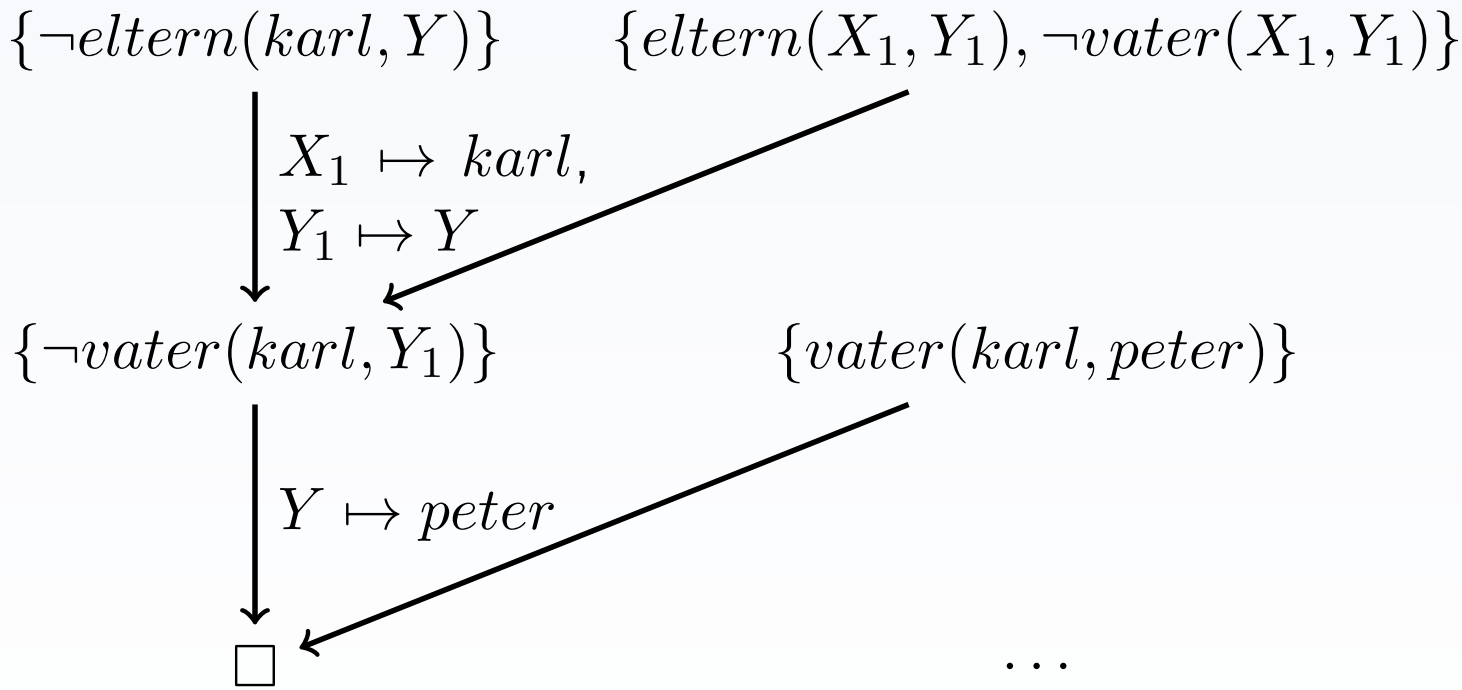
- Zeige: $\mathcal{F} \models G$ mit Resolution: $\mathcal{F}, \neg G$ ist widersprüchlich
- $\neg\exists Y.eltern(karl, Y)$ wird zur Anfrageklausel $\{\neg eltern(karl, Y)\}$
- Verwende lineare Resolution



Beispiel (2)

Anfrage / Aufruf: $\exists Y.eltern(karl, Y)$ (hat Karl Kinder?)

- Zeige: $\mathcal{F} \models G$ mit Resolution: $\mathcal{F}, \neg G$ ist widersprüchlich
- $\neg\exists Y.eltern(karl, Y)$ wird zur Anfrageklausel $\{\neg eltern(karl, Y)\}$
- Verwende lineare Resolution

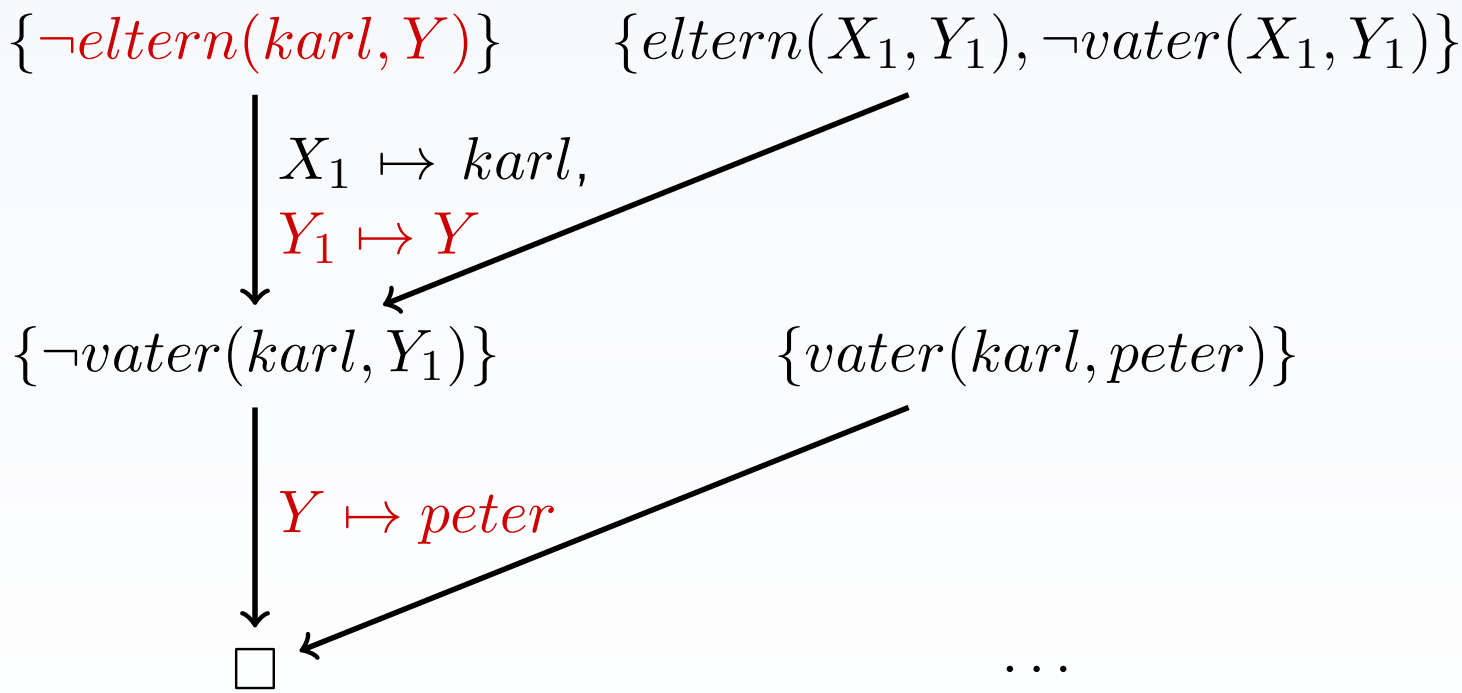


Welche Kinder hat Karl? Wo ist die Antwort?

Beispiel (2)

Anfrage / Aufruf: $\exists Y.eltern(karl, Y)$ (hat Karl Kinder?)

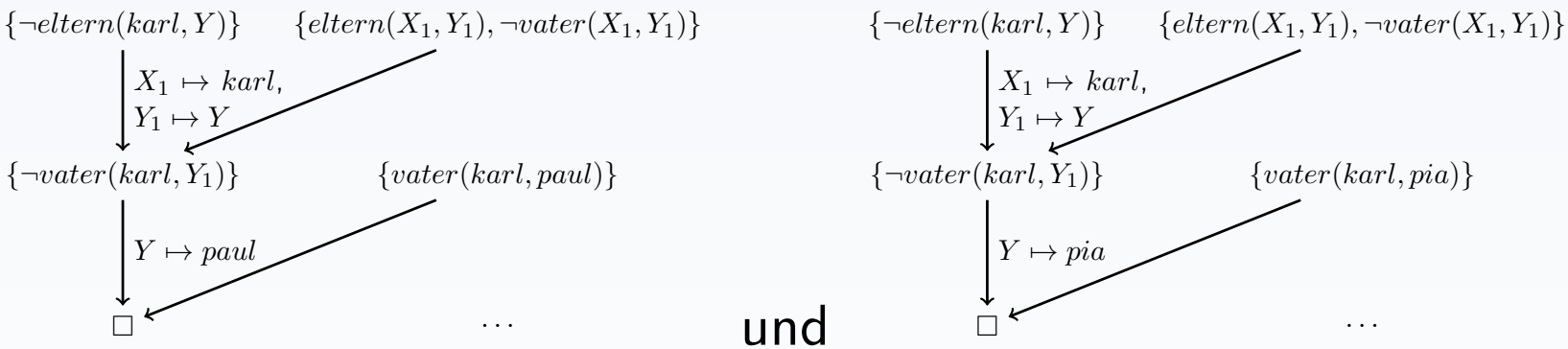
- Zeige: $\mathcal{F} \models G$ mit Resolution: $\mathcal{F}, \neg G$ ist widersprüchlich
- $\neg\exists Y.eltern(karl, Y)$ wird zur Anfrageklausel $\{\neg eltern(karl, Y)\}$
- Verwende lineare Resolution



Welche Kinder hat Karl? Wo ist die Antwort?

Beispiel (3)

Es gibt noch zwei weitere Kinder:



Dies sind gerade alle Möglichkeiten durch lineare Resolution die leere Klausel herzuleiten.

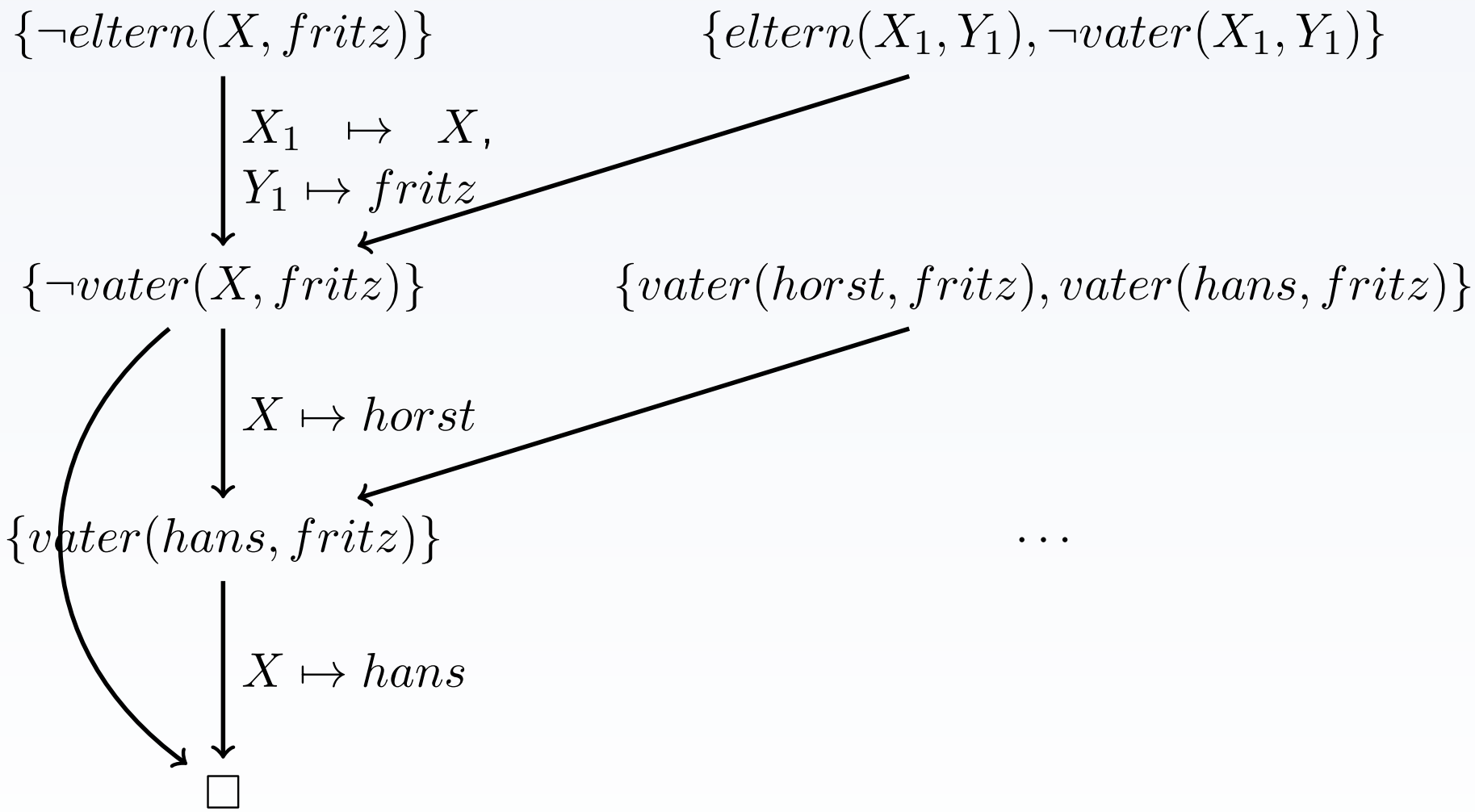
Antwort erzeugen

- Verwende die entstandenen Unifikatoren
  um die Anfrageklausel zu instanziieren
- Funktioniert das immer?

Beispiel:

- Füge die Aussage hinzu:
  „Der Vater von Fritz ist Horst oder Hans“
- Klausel $\{vater(horst, fritz), vater(hans, fritz)\}$
- Anfrage: $\exists X : eltern(X, fritz)$,

Antwort eindeutig?



Die Antwort ist nicht eindeutig!

Eindeutige Antworten

Mehrdeutige Antworten treten auf wenn

- Eingabeklauseln erhalten mehr als ein positives Literal
  $\{vater(horst, fritz), vater(hans, fritz)\}$
- Oder: Anfrage zerfällt in mehrere Zielklauseln:
  - Anfrage ist Disjunktion:
  - $\exists X_1, X_2 : eltern(X_1, maria) \vee eltern(X_2, monika)$
  - CNF: $\{\neg eltern(X_1, maria)\}$ und $\{\neg eltern(X_2, monika)\}$

Eindeutige Antworten (2)

Konsequenz: Funktioniert nicht mit allen Klauseln

Aber: Mit **Hornklauseln** und **einer** Zielklausel funktioniert es!
Vorteile von Hornklauseln

- Hornklauseln enthalten höchstens ein positives Literal
- Viele Zusammenhänge sind mit Hornklauseln noch ausdrückbar
- SLD-Resolution ist korrekt und vollständig

Hornklauseln

- **Hornklausel**: **maximal ein positives Literal**.
- **definite Klausel**: **genau ein positives Literal**
  D.h. $A \vee \neg B_1 \vee \ldots \vee \neg B_m$
  Entspricht gerade $B_1 \wedge \ldots \wedge B_m \implies A$

  Notation in der logischen Programmierung:

  $$A \Leftarrow B_1, \ldots, B_m.$$

  Beachte: Kommata sind Konjuktionen,Punkt legt das Ende der Klausel fest.
  Prolog-Schreibweise (:- entspricht $\Leftarrow$):

  $$A \text{ :- } B_1, \ldots, B_m.$$

  Sprechweisen:
  - $A =$ Kopf
  - $B_1, \ldots, B_m =$ Rumpf

Hornklauseln (2)

- **Fakt**: Definite Klausel mit leerem Rumpf
  (1-Klausel mit positivem Literal)

  Notation in der logischen Programmierung: $A \Leftarrow$.

  In Prolog: A.

- **Definites Ziel** (*Anfrage*,*Query*,*goal*):
  Klausel **ohne positive** Literale: $\neg B_1 \vee \ldots \vee \neg B_n$

  Notation in der logischen Programmierung:

  $$\Leftarrow B_1, \ldots, B_n.$$

  Die B_i werden Unterziele (Subgoals) genannt.

  Im Prolog-Interpreter: Man gibt $B_1, \ldots, B_n.$ ein.
  Mit dem Prompt ?- $B_1, \ldots, B_n.$

Hornklauseln (3)

- **Definites Programm**: Menge von definiten Klauseln.
- **Definition von Q**: Menge aller Klauseln, deren Kopfliteral das Prädikat Q hat
  Bsp.: Definition von $eltern$

Hornklauseln (3)

- **Definites Programm**: Menge von definiten Klauseln.
- **Definition von Q**: Menge aller Klauseln, deren Kopfliteral das Prädikat Q hat
  Bsp.: Definition von $eltern$

  $\{eltern(X, Y), \neg vater(X, Y)\}$
  $\{eltern(X, Y), \neg mutter(X, Y)\}$

Hornklauseln (3)

- **Definites Programm**: Menge von definiten Klauseln.
- **Definition von Q**: Menge aller Klauseln, deren Kopfliteral das Prädikat Q hat
  Bsp.: Definition von $eltern$

  $\{eltern(X, Y), \neg vater(X, Y)\}$
  $\{eltern(X, Y), \neg mutter(X, Y)\}$

  Notation in der logischen Programmierung:
  $eltern(X, Y) \Leftarrow vater(X, Y).$
  $eltern(X, Y) \Leftarrow mutter(X, Y).$

Hornklauseln (3)

- **Definites Programm**: Menge von definiten Klauseln.
- **Definition von Q**: Menge aller Klauseln, deren Kopfliteral das Prädikat Q hat
  Bsp.: Definition von $eltern$

  $\{eltern(X, Y), \neg vater(X, Y)\}$
  $\{eltern(X, Y), \neg mutter(X, Y)\}$

  Notation in der logischen Programmierung:
  $eltern(X, Y) \Leftarrow vater(X, Y).$
  $eltern(X, Y) \Leftarrow mutter(X, Y).$

  In Prolog:
  $eltern(X, Y)$:- $vater(X, Y).$
  $eltern(X, Y)$:- $mutter(X, Y).$

Prolog-Programm zum Beispiel

```
vater(peter,maria).
vater(peter,monika).
vater(karl, peter).
vater(karl, pia).
vater(karl, paul).
mutter(susanne,monika).
mutter(susanne,maria).
mutter(elisabeth, peter).
mutter(elisabeth, pia).
mutter(elisabeth, paul).

eltern(X,Y) :-  vater(X,Y).
eltern(X,Y) :-  mutter(X,Y).
```

Prolog

Prolog = Programmation en Logique

Prolog-Interpreter (Auswahl)

- SWI-Prolog: http://www.swi-prolog.org/
  Freie Software, ISO-kompatibel, mit Editor, Debugger
- SICStus Prolog, kommerziell
  entwickelt vom Swedish Institute of Computer Science
  Personal license: 165 EUR,
  kommerzielle single-user Lizenz: 2100 EUR
- GNU Prolog: http://www.gprolog.org
  ISO-kompatibel, quelloffen
- ECLiPSe Constraint Programming System
  Prolog-basiert mit Erweiterungen zur Constraint-basierten Programmierung
  frei, quelloffen

Wir verwenden SWI-Prolog.

SWI-Prolog

Laden der Datei in den Interpreter:

- Dateiname: verwandte.pl
- Im Interpreter consult('verwandte').

Anfragen

- Werden direkt am Prompt eingegeben (Beenden mit Punkt)
- Weitere Lösungen finden: Semikolon

Beispiele

```
?- eltern(karl,X).
X = peter ;
X = pia ;
X = paul ;
false.

?- eltern(karl,x).
false.

?- eltern(peter,X).
X = maria ;
X = monika ;
false.

?- eltern(X,peter).
X = karl ;
X = elisabeth.
```

Beispiele

```
?- eltern(X,Y).
X = peter,
Y = maria ;
X = peter,
Y = monika ;
X = karl,
Y = peter ;
X = karl,
Y = pia ;
X = karl,
Y = paul ;
X = susanne,
Y = monika ;
...
```

Syntaxkonventionen in Prolog

- Variablen: beginnen mit Großbuchstaben
- Prädikate und Funktionssymbole: beginnen mit Kleinbuchstaben

Daher liefert die Anfrage

```
?- eltern(karl,x).
```

kein Ergebnis (`x` wird als Konstante interpretiert!).

Semantik von Hornklauselprogrammen

- Zunächst die allgemeine (operationale) Semantik von Hornklauselprogrammen
- Diese ist nichtdeterministsch
- Beachte: Prolog verwendet veränderte (deterministische) Semantik
- Daher: Prolog-Semantik im Anschluss

Semantik von Hornklauselprogrammen (2)

Im Wesentlichen

- SLD-Resolution
- mit Antworterzeugung

SLD-Resolution:

- ein definites Ziel
- jede Resolution verwendet
  - Resolvente (Zentralklausel)
  - Seitenklausel
- Umbenennung: Es reicht aus stets die Seitenklausel umzubenennen
- Mit dieser Umbenennung: Standardisierte SLD-Resolution

Standardisierte SLD-Resolution

Sei $G = \Leftarrow A_1, \ldots, A_m, \ldots, A_k$ definites Ziel und $C = A \Leftarrow B_1, \ldots, B_q$ definite Klausel, wobei C frisch umbenannt ist. Leite aus G und C mit Resolution neues Ziel G' her:

1. A_m ist das durch die Selektionsfunktion selektierte Atom von G.
2. θ ist ein allgemeinster Unifikator von A_m und A, dem Kopf von C.
3. G' ist das neue Ziel: $\theta(A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)$.

Ableitungsrelation: $G \rightarrow_{\theta,C,m} G'$

Logische Programmierung:

$$\frac{\Leftarrow A_1, \ldots, A_m, \ldots, A_k \qquad A \Leftarrow B_1, \ldots, B_q}{\sigma(A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)} \quad \sigma(A_m) = \sigma(A)$$

PL-Darstellung

$$\frac{\neg A_1 \vee \cdots \vee \neg A_m \vee \cdots \vee \neg A_k \qquad A \vee \neg B_1 \vee \cdots \vee \neg B_q}{\sigma(A_1 \vee \cdots \vee A_{m-1} \vee B_1 \vee \cdots \vee B_q \vee A_{m+1} \vee \cdots \vee A_k)} \quad \sigma(A_m) = \sigma(A)$$

SLD-Ableitung

Sei P definites Programm und G definites Ziel.
Eine **SLD-Ableitung** von $P \cup \{G\}$ ist eine Folge

$$G \rightarrow_{\theta_1, C_1, m_1} G_1 \rightarrow_{\theta_2, C_2, m_2} G_2 \ldots$$

von SLD-Schritten, wobei C_i jeweils eine umbenannte Klausel aus P ist

Semantik von Hornklauselprogrammen (2)

- **SLD-Widerlegung**: SLD-Ableitung, die mit $\square$ endet.
- **Erfolgreiche SLD-Ableitung** = SLD-Widerlegung
- **fehlgeschlagene SLD-Ableitung**: Wenn diese nicht fortsetzbar ist, und nicht mit $\square$ endet
- **unendliche SLD-Ableitung**: $G \to G_1 \to G_2 \to \ldots$

Antworten

Sei P ein definites Programm und G ein definites Ziel.

- **Korrekte Antwort**: Substitution θ, so dass $P \models \theta(\neg G)$ gilt.
- **Berechnete Antwort**: Substitution θ mit
  Erfolgreiche SLD-Ableitung $P \cup \{G\}$

$$G \rightarrow_{\theta_1, C_1, m_1} G_1 \rightarrow_{\theta_2, C_2, m_2} G_2 \ldots \rightarrow_{\theta_n, C_n, m_n} \square$$

θ ist die Komposition $\theta_n \circ \ldots \circ \theta_1$, wobei man diese auf die Variablen von G einschränkt.

Korrektheit

Soundness der SLD-Resolution

Sei P ein definites Programm und G ein definites Ziel.
Dann ist jede **berechnete** Antwort θ auch **korrekt**. D.h.
$P \models \theta(\neg G)$

Widerlegungsvollständigkeit

Widerlegungsvollständigkeit

Sei P ein definites Programm und G ein definites Ziel.
Wenn $P \cup \{G\}$ unerfüllbar ist, dann gibt es eine SLD-Widerlegung von $P \cup \{G\}$.

Vollständigkeit

Es gilt ein stärkerer Satz:

Vollständigkeit der SLD-Resolution

Sei P ein definites Programm und G ein definites Ziel.
Zu jeder **korrekten** Antwort θ gibt es eine **berechnete** Antwort σ für $P \cup \{G\}$ und eine Substitution γ so dass für alle Variablen $x \in FV(G)$: $\gamma\sigma(x) = \theta(x)$.

D.h. die berechnete Antwort ist eine **allgemeinste Antwort**!

Algorithmus zur Berechnung aller Antworten

Algorithmus **Berechnung aller Antworten**

1. Gegeben ein Ziel $\Leftarrow A_1, \ldots, A_n$:
   1. Probiere alle Möglichkeiten aus, ein Unterziel A aus $A_1, \ldots, A_n$ auszuwählen
   2. Probiere alle Möglichkeiten aus, eine Resolution von A mit einem Kopf einer Programmklausel durchzuführen.
2. Erzeuge neues Ziel B: Löschen von A, Instanziieren des restlichen Ziels, Hinzufügen des Rumpfs der Klausel.
3. Wenn $B = \square$, dann gebe die Antwort aus. Sonst: mache weiter mit 1. mit dem Ziel B.

Eigenschaften

Dieser Algorithmus hat zwei Verzweigungspunkte pro Resolutionsschritt:

- Die Auswahl eines Atoms
- Die Auswahl einer Klausel

Aber: Auswahl des Atoms ist don't care, denn:

Satz

Vertauscht man in einer SLD-Widerlegung die Abarbeitung zweier Atome in einem Ziel, so sind die zugehörigen Substitutionen bis auf Variablenumbenennung gleich und die Widerlegung hat die gleiche Länge.
Weiterhin: die Suchstrategie, die irgendein Atom auswählt, dann alle Klauseln durchprobiert, usw. ist vollständig bzgl der Antworten.

Achtung: Betrifft nicht die Reihenfolge, in der Seitenklauseln ausgewählt werden, diese ist nach wie vor nichtdeterministisch.

Eigenschaften (2)

Man kann daher leicht zu lösende Atome zuerst auswählen und schwerer zu lösende zurückstellen.

Aber: Es kann aber sein, dass bei günstiger Auswahl nur endlich viele Alternativen ausprobiert werden müssen, aber bei ungünstiger Auswahl evtl. eine unendliche Ableitung ohne Lösungen mitbetrachtet werden muss.

(Beispiel folgt gleich)

SLD-Bäume

Gegeben ein definites Programm P, und ein definites Ziel G: Ein **SLD-Baum** für $P \cup \{G\}$ ist ein Baum der folgendes erfüllt:

1. Jeder Knoten ist markiert mit einem definiten Ziel
2. die Wurzel ist markiert mit G
3. In jedem Knoten mit nichtleerem Ziel wird ein Atom A des Ziels mit der Selektionsfunktion ausgewählt. Die Kinder dieses Knotens sind dann die möglichen Ziele nach genau einem SLD-Resolutionsschritt mit einer definiten Klausel in P.
4. Knoten, die mit der leeren Klausel markiert sind, sind Blätter.
5. Ein Blatt ist entweder mit dem leeren Ziel markiert, oder es gibt von dem Blatt aus keine SLD-Resolution.

Beispiel

Programm:

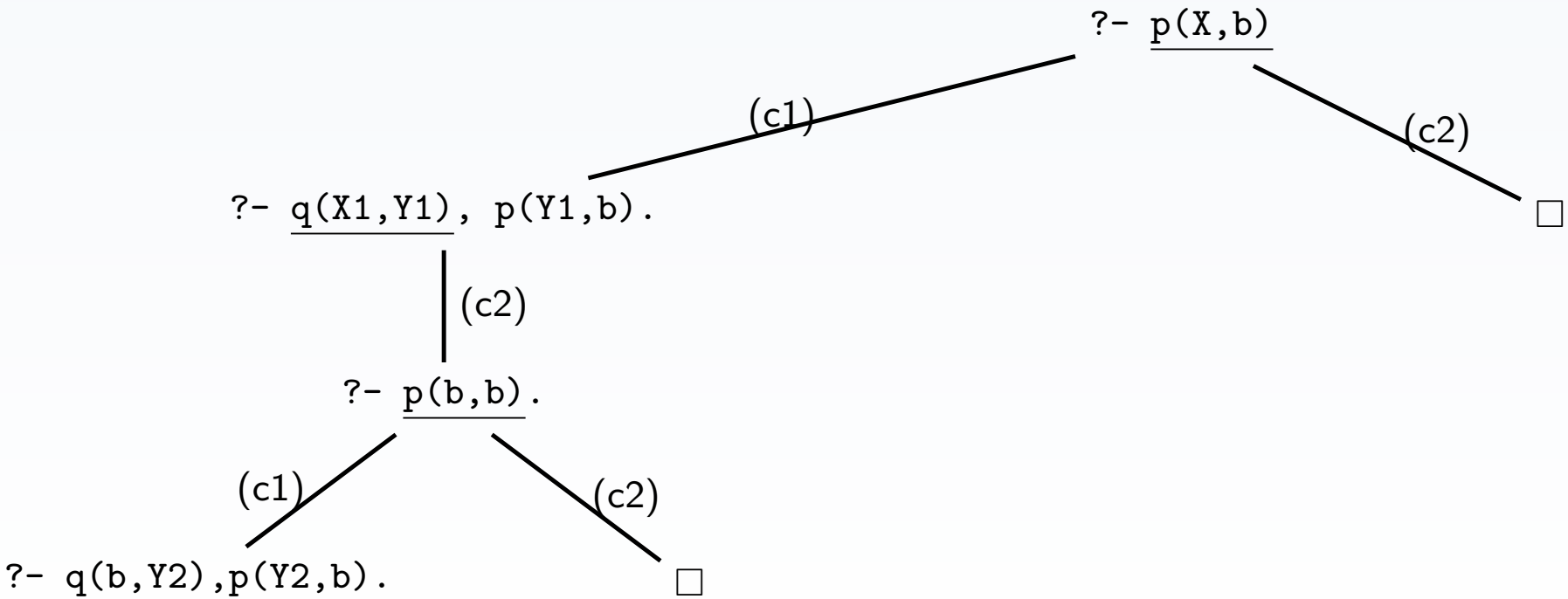
```
(c1)        p(X,Z) :-  q(X,Y), p(Y,Z).
(c2)        p(X,X).
(c3)        q(a,b).
```

Anfrage: `?- p(X,b)` (Zur Erinnerung: entspricht Klausel $\{\neg p(X, b)\}$)

Beispiel (2)

```
(c1)        p(X,Z) :-  q(X,Y), p(Y,Z).
(c2)        p(X,X).
(c3)        q(a,b).
(a)         ?-p(X,b)
```

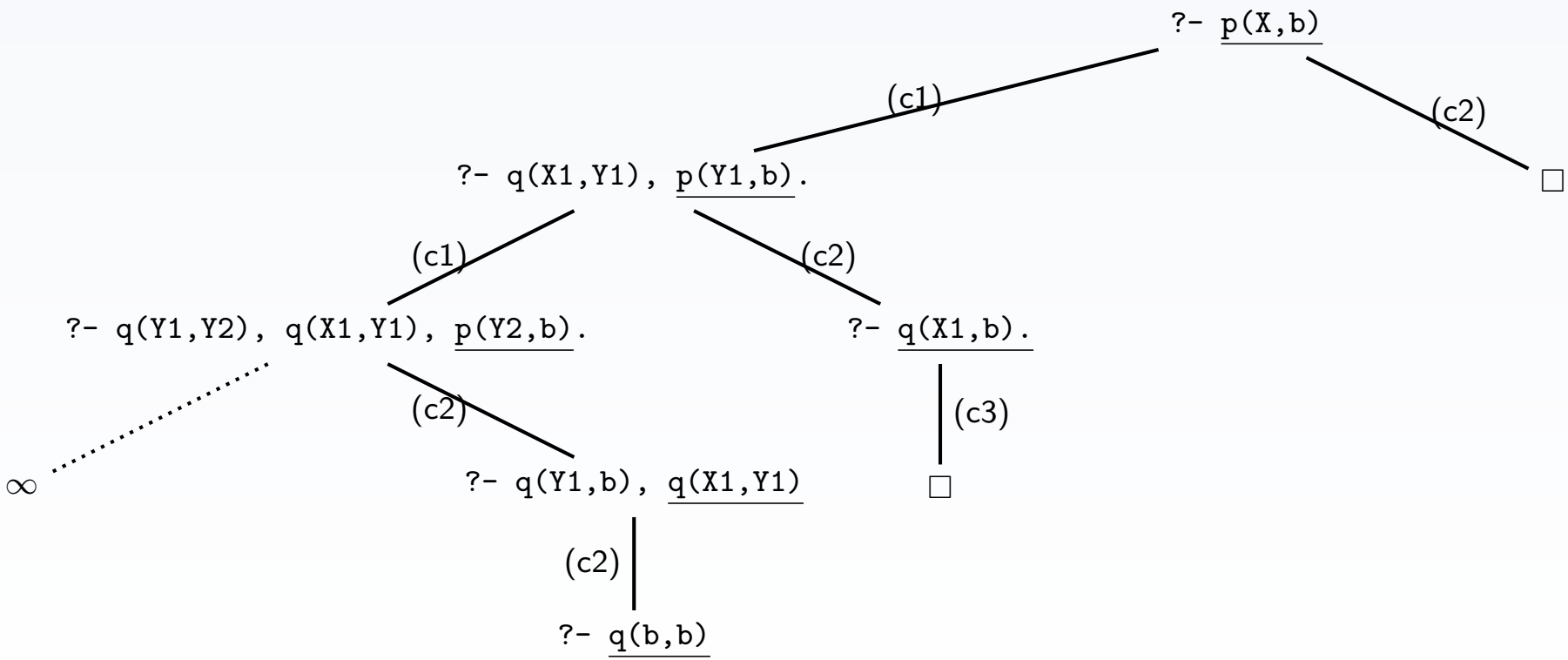
Selektionsfunktion: Erst q dann p:



Beispiel (3)

```
(c1)        p(X,Z) :-  q(X,Y), p(Y,Z).
(c2)        p(X,X).
(c3)        q(a,b).
(a)         ?-p(X,b)
```

Selektionsfunktion: Erst p dann q:



Eigenschaften (3)

Beispiel zeigt:

- Bei ungünstiger Wahl der Selektionsfunktion gibt es unendliche Pfade

Aber

- Breitensuche findet alle Lösungen nach endlich vielen Schritten
- Tiefensuche findet nicht alle Lösungen!
- Breitensuche allerdings sehr platzintensiv!

Syntaxkonventionen in Prolog

- **Namen** bestehen aus Buchstaben, Ziffen und _
- **Konstanten, Prädikate, Funktionssymbole**: Namen die mit Kleinbuchstaben beginnen
- **Variablen**: Namen die mit Großbuchstaben oder _ beginnen.
- **Wildcard**: _ alleine (Variable ohne Namen)

Suche / Resolution in Prolog

- Anfrage als Stack von Literalen, das oberste Literal wird zuerst bearbeitet
- Programmklauseln werden in der Reihenfolge abgesucht, in der sie im Programm stehen.
- SLD-Resolutionsschritt ersetzt Literal durch den Rumpf

Eigenschaften

- deterministisch
- Tiefensuche
- Nicht vollständig!

Begründungen

Warum nimmt man dieses nicht vollständige Verfahren?

- Oft schneller
- benötigt weniger Platz
- Programmierer ist verantwortlich die Klauseln „terminierend“ anzuordnen.

Beachte:

- Entspricht eigentlich nicht dem Paradigma des deklarativen Programmierens
- Denn: Programmierer spezifiziert auch das Wie nicht nur das Was

Beispiele

`vorfahr(X,Y)` soll wahr sein, wenn `X` ein Vorfahre von `Y` ist.

```
vorfahr1(X,Z) :- vorfahr1(X,Y),vorfahr1(Y,Z).
vorfahr1(X,Y) :- eltern(X,Y).
```

Terminiert so nicht!

Beispiele (2)

`vorfahr(X,Y)` soll wahr sein, wenn `X` ein Vorfahre von `Y` ist.

```
vorfahr2(X,Y) :- eltern(X,Y).
vorfahr2(X,Z) :- vorfahr2(X,Y),vorfahr2(Y,Z).
```

```
vorfahr(X,Y) :- eltern(X,Y).
vorfahr(X,Z) :- eltern(X,Y),vorfahr(Y,Z).
```

Terminiert!

Anordnen der Klauseln zur Terminierung

Beispiel, das zeigt: Anordnen funktioniert nicht immer:

```
p(a,b).
p(c,b).
p(X,Y) :- p(Y,X).
p(X,Z) :- p(X,Y),p(Y,Z).
```

Unabhängig davon, wie man die Klauseln anordnet:

- Anfrage `p(a,c)` terminiert nicht
- Aber: Es gibt eine erfolgreiche SLD-Resolution

Mögliche Abhilfe

Einführen eines Parameters zur Beschränkung der Tiefe:

```
p(_,a,b).
p(_,c,b).
p(I,X,Y) :-  I > 0, J is I-1,  p(J,Y,X).
p(I,X,Z) :-  I > 0, J is I-1,  p(J,X,Y) , p(J,Y,Z).
```

Ruft man `p(2,a,c)` auf so erhält man true.

Iteratives Vertiefen

```
startp(X,Y) :- itp(0,X,Y).

itp(I,X,Y) :- p(I,X,Y).
itp(I,X,Y) :- itp(I+1,X,Y).

p(_,a,b).
p(_,c,b).
p(I,X,Y) :-  I > 0, J is I-1,  p(J,Y,X).
p(I,X,Z) :-  I > 0, J is I-1,  p(J,X,Y) , p(J,Y,Z).
```

Unifikation in Prolog

Prolog kann Unifizieren:

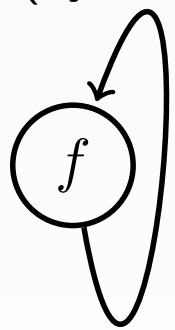
```
?- h(a,X,g(Z,f(b))) = h(Y,g(f(Y),W),U) .
X = g(f(a), W),
Y = a,
U = g(Z, f(b)).
```

Unifikation in Prolog (2)

Aber: Viele Implementierungen führen den Occurs-Check **nicht** durch:

```
?- f(X) = f(f(X)).
X = f(**).
```

X wird als unendlicher Term (zyklischer Graph) aufgefasst:



Entspricht **nicht** der PL_1-Semantik!

Weitere Veränderungen in Prolog gegenüber der Theorie

- Es gibt extra Operatoren zur Steuerung des Backtracking: `cut`
- Negation: Negation wird definiert als Fehlschlagen der Suche.
- Es wird z.T. Typinformation zu Programmen hinzugefügt.
- Spezialprädikate für Ein-/Ausgabe
- Arithmetische Operationen und Zahlen werden extra hinzugefügt
- `assert`/`retract`: Erlaubt das dynamische Hinzufügen / Löschen von Programmklausen (i.A. nur Fakten).

Beispiel zu assert/retract

```
?- assert(tier(hund)).
true.

?- assert(tier(katze)).
true.

?- tier(X).
X = hund ;
X = katze.

?- retract(tier(hund)).
true.

?- tier(X).
X = katze.

?- retract(tier(katze)).
true.

?- tier(X).
false.
```

Prolog: Arithmetische Operationen

- Operatoren: $+, -, *, /$ dürfen infix verwendet werden
- Ausdrücke wie $a * b + c$ Abkürzung für Terme, $+(*(a, b), c)$.
- Zahlen sind erlaubt und als Standard in Prolog implementiert.
- Das Spezialprädikat $=$ (Gleichheit) ist vordefiniert als:

  `X = X.`

Prolog: Arithmetische Operationen

- Operatoren: $+, -, *, /$ dürfen infix verwendet werden
- Ausdrücke wie $a * b + c$ Abkürzung für Terme, $+(*(a, b), c)$.
- Zahlen sind erlaubt und als Standard in Prolog implementiert.
- Das Spezialprädikat $=$ (Gleichheit) ist vordefiniert als:

  `X = X.`
- Das ist jedoch nur **syntaktische Gleichheit** (mit Unifikation)

```
?- 3+4 = 7.
false.

?- 3+X = Y+4.
X = 4,
Y = 3.

?- 2*(3+4) = Z*W.
Z = 2,
W = 3+4.
```

Vergleiche

`=`	Syntaktische Gleichheit (Unifikation erlaubt)
`=:=`	Wertgleichheit, die Argumente werden zu Zahlen ausgewertet und dürfen keine Variablen enthalten
`=/=`	Wertungleichheit
`<`	kleiner als Wertgleichheit
`>`	größer als Wertgleichheit
`=<`	kleiner gleich als Wertgleichheit
`>=`	größer gleich als Wertgleichheit

Bei Wertgleichheiten:

Wenn exp1 op exp2 ausgewertet wird,
müssen exp1 und exp2 zu einer Zahl auswertbar sein.

Beispiele

```
?- 3+4 = X.
X = 3+4.

?- 3+4 =:= 7.
true.

?- 3+4 =:= X.
ERROR: =:=/2: Arguments are not sufficiently instantiated
?- X < 6.
ERROR: </2: Arguments are not sufficiently instantiated
?- 5 < 6.
true.

?- (2*2) < 6.
true.
```

Wertzuweisung

Mit den bisherigen Operationen nicht möglich: Instanziiere Variable mit Wert.

Spezialprädikat `is`:

Variable `is` arithmetischer Ausdruck

Semantik: Werte Ausdruck aus und binde den Wert an die Variable

```
?- X is (2+2*4).
X = 10.

?-  (2+2*4) is Y.
ERROR: is/2: Arguments are not sufficiently instantiated
?- X is Y.
ERROR: is/2: Arguments are not sufficiently instantiated
?- X=2, Y is X.
X = 2,
Y = 2.
```

Semantik

Beachte: Alle Prolog-Prädikate die Werte ausrechnen, passen nicht direkt zur PL_1-Semantik!

Listen

Listen: Darstellung als Terme

- Konstante `[]` (gesprochen „Nil“) für die leere Liste
- Zweistelligen Funktionssymbol „Cons“: Erhält Listenelement und Restliste
  In Prolog ist dieses Funktionssymbol als Punkt `.` dargestellt.

Liste `[1,2,3]` wird als `.(1,.(2,.(3,[])))` dargestellt
Prolog erlaubt aber auch: Schreibweise `[1,2,3]` als syntaktischen Zucker

```
?- [1,2,3]   = .(1,.(2,.(3,[]))).
true.
?- [1,2,3]   = .(1,.(2,.(3,.(4,[])))).
false.
?- [1,2,3,4] = .(1,.(2,.(3,.(4,[])))).
true.
```

Listen (2)

Listen in Prolog sind heterogen: Beliebige Elemente erlaubt

```
 ?- [[1], [2,3], [4,5]] = .(.(1,[]), .(.(2,.(3,[])),.(.(4,.(5,[])),[]))).
true.
?- [1,[1],[[1]],[[[1]]]] = .(1,.(.(1,[]),.(.(.(1,[]),[]),.( .(.(.(1,[]),[]),[]),[])))).
true.
?- [f(g(a)),h(b,x),f(f(f(f(c))))] = .(X,.(Y,.(Z,[]))).
X = f(g(a)),
Y = h(b, x),
Z = f(f(f(f(c)))).
```

Listenprogrammierung

Selektoren:

```
head(.(X,_),X).
tail(.(_,XS),XS).
```

Tests:

```
?- head([1,2,3,4],X).
X = 1.

?- tail([1,2,3,4],X).
X = [2, 3, 4].
```

Listenprogrammierung (2)

- Prolog: bietet komfortablere Schreibweise für Listenpattern
- `.(x,xs)` kann als `[X|XS]` geschrieben werden.
- `|` ist noch allgemeiner: Zerlegen an einer beliebigen Position: auch `[X,Y,Z|YS]` erlaubt

head und tail mit dieser Syntax:

```
head([X|_],X).
tail([_|XS],XS).
```

Einige Aufrufe:

```
?- head([1,2,3],X).
X = 1.

?- tail([1,2,3],X).
X = [2, 3].

?- [X,Y,Z|ZS] = [1,2,3,4,5].
X = 1,
Y = 2,
Z = 3,
ZS = [4, 5].
```

Listenprogrammierung (3)

Beispiele mit Variablen:

```
?- head(X,Y).
X = [Y|_G304].

?- tail(X,Y).
X = [_G303|Y].
```

Listenprogrammierung: member

member(E,XS): wahr wenn E Element der Liste XS ist

```
member(X,[X|_]).
member(X,[_|Y]) :-  member(X,Y).
```

Einige Beispielaufrufe:

```
?- member(2,[1,2,3]).
true.

?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
false.

?- member(X,Y).
Y = [X|_G304].
```

Listenprogrammierung: member (2)

```
member(X,[X|_]).
member(X,[_|Y]) :-  member(X,Y).
```

Terminiert `member` stets?

- `member(s,t)`: wenn t keine Variablen enthält. dann wird t' beim nächsten mal kleiner.
- `member(Y,Y)`: Die erste Klausel unifiziert wenn occurs-check aus ist
  Mit zweiter Klausel:

  ```
  member(X_1,[_|Y_1]) :-  member(X_1,Y_1).
  X_1 = Y = [_|Y_1]
  ```

  Die neue Anfrage ist: `member([_|Y_1] ,Y_1)`

  Die nächste Unifikation ebenfalls mit zweiter Klausel

  ```
  member(X_2,[_|Y_2]) :-  member(X_2,Y_2)
  X_2 = [_|Y_1] , Y_1 = [_|Y_2]
  ```

  usw. Man sieht: das terminiert nicht.

Listenprogrammierung: member (3)

Testen in Prolog: member wie vorher, member2: Programmklauseln umgekehrt

```
?- member2(X,[1,2,3]).
X = 3 ;
X = 2 ;
X = 1.

?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3.

?- member(Y,Y).
Y = [**|_G460]

?- member2(Y,Y).
^C... terminiert nicht
```

Listenprogrammierung: islist

```
islist([_|X]) :- islist(X).
islist([]).
```

terminiert für Listen ohne Variablen, aber nicht für `islist(X)`.
umgekehrt:

```
islist([]).
islist([_|X]) :- islist(X).
```

terminiert für die Anfrage `islist(X)`.

Listenprogrammierung: laenge

```
laenge([],0).
laenge([_|X],N) :- laenge(X,N_1), N is N_1 + 1.
```

Beispiele:

```
?- laenge([1,2,3],N).
N = 3.

?- laenge(XS,2).
XS = [_G880, _G883] ;
^C
?- laenge(XS,N).
XS = [],
N = 0 ;
XS = [_G892],
N = 1 ;
XS = [_G892, _G895],
N = 2 ;
...
```

Listenprogrammierung: Sortieren

```
% Praedikat, das testet ob eine Liste sortiert ist

sortiert([]).
sortiert([X]).
sortiert([X|[Y|Z]]) :- X =< Y, sortiert([Y|Z]).

% sortiert_einfuegen(A,BS,CS):
%   fuegt A sortiert in BS ein, Ergebnis: CS

sortiert_einfuegen(X,[],[X]).
sortiert_einfuegen(X,[Y|Z], [X|[Y|Z]]) :-  X =< Y.
sortiert_einfuegen(X,[Y|Z], [Y|U]) :-
  Y < X, sortiert_einfuegen(X,Z,U).

% ins_sortiere sortiert die Liste

ins_sortiere(X,X) :-  sortiert(X).
ins_sortiere([X|Y], Z) :-
  ins_sortiere(Y,U), sortiert_einfuegen(X,U,Z).
```

Listenprogrammierung: Sortieren (2)

Einige Testaufrufe:

```
?- sortiert([1,2,3]).
true.
?- sortiert([2,3,1]).
false.
?- sortiert([1,2,X]).
ERROR: =</2: Arguments are not sufficiently instantiated

?- sortiert(X).
X = [] ;
X = [_G312] ;
ERROR: =</2: Arguments are not sufficiently instantiated


?- ins_sortiere([5,1,4,2,3],X).
X = [1, 2, 3, 4, 5] ;
X = [1, 2, 3, 4, 5] ;
X = [1, 2, 3, 4, 5] ;
false.
```

Listenprogrammierung: append

```
% append(A,B,C) haengt Listen A und B zusammen

append([],X,X).
append([X|Y],U,[X|Z]) :- append(Y,U,Z).
```

Beispiele:

```
?- append([1,2],[3,4],Z).
Z = [1, 2, 3, 4].

?- append([1,2],Y,Z).
Z = [1, 2|Y].

?- append(X,Y,Z).
X = [],
Y = Z ;
X = [_G663],
Z = [_G663|Y] ;
...
```

Listenprogrammierung: merge

```
merge([],YS,YS).
merge(XS,[],XS).
merge([X|XS], [Y|YS], [X|ZS]) :- merge(XS,[Y|YS],ZS), X =< Y.
merge([X|XS], [Y|YS], [Y|ZS]) :- merge([X|XS],YS,ZS), X > Y.
```

Beispiele:

```
?- merge([1,10,100],[0,5,50,500],Z).
Z = [0, 1, 5, 10, 50, 100, 500].

?- merge(X,Y,[1,2,3]).
X = [],
Y = [1, 2, 3] ;
X = [1, 2, 3],
Y = [] ;
X = [1],
Y = [2, 3] ;
...
```

Weitere Listenfunktionen

```
listtoset([],[]).
listtoset([X|R],[X|S]) :- remove(X,R,S1),listtoset(S1,S).

remove(E,[],[]).
remove(E,[E|R],S):- remove(E,R,S).
remove(E,[X|R],[X|S]):- not(E == X), remove(E,R,S).

union(X,Y,Z) :- append(X,Y,XY), listtoset(XY,Z).

intersect([],X,[]).
intersect([X|R],S,[X|T]) :- member(X,S),
                            intersect(R,S,T1),
listtoset(T1,T).
intersect([X|R],S,T) :-     not(member(X,S)),
                            intersect(R,S,T1),
listtoset(T1,T).
```

Weitere Listenfunktionen (2)

```
reverse([],[]).
reverse([X|R],Y) :- reverse(R,RR), append(RR,[X],Y).

reversea(X,Y) :- reverseah(X,[],Y).
reverseah([X|Xs],Acc,Y):- reverseah(Xs,[X|Acc],Y).
reverseah([],Y,Y).
```

Differenzlisten als effiziente Implementierung

Betrachte erneut append:

```
append([],X,X).
append([X|Y],U,[X|Z]) :- append(Y,U,Z).
```

Nachteil: Laufzeit ist linear in der ersten Liste.

Abhilfe: Differenzlisten

Andere Darstellung von Listen:

Liste wird als Differenz $L_1 - L_2$ dargestellt, wobei

- L_1 und L_2 sind Listen
- L_2 ist Suffix von L_1
- Die eigentliche Liste ist L_1 ohne den Suffix L_2

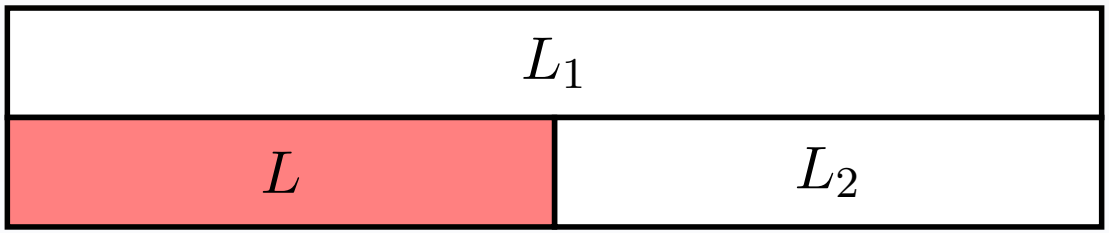
Beispiel `[1,2,3]`:

- `[1,2,3,4,5] - [4,5]` oder
- `[1,2,3,4,5,6] - [4,5,6]` oder ...
- Allgemein `[1,2,3|Y] - Y`

Darstellung ist **nicht eindeutig**

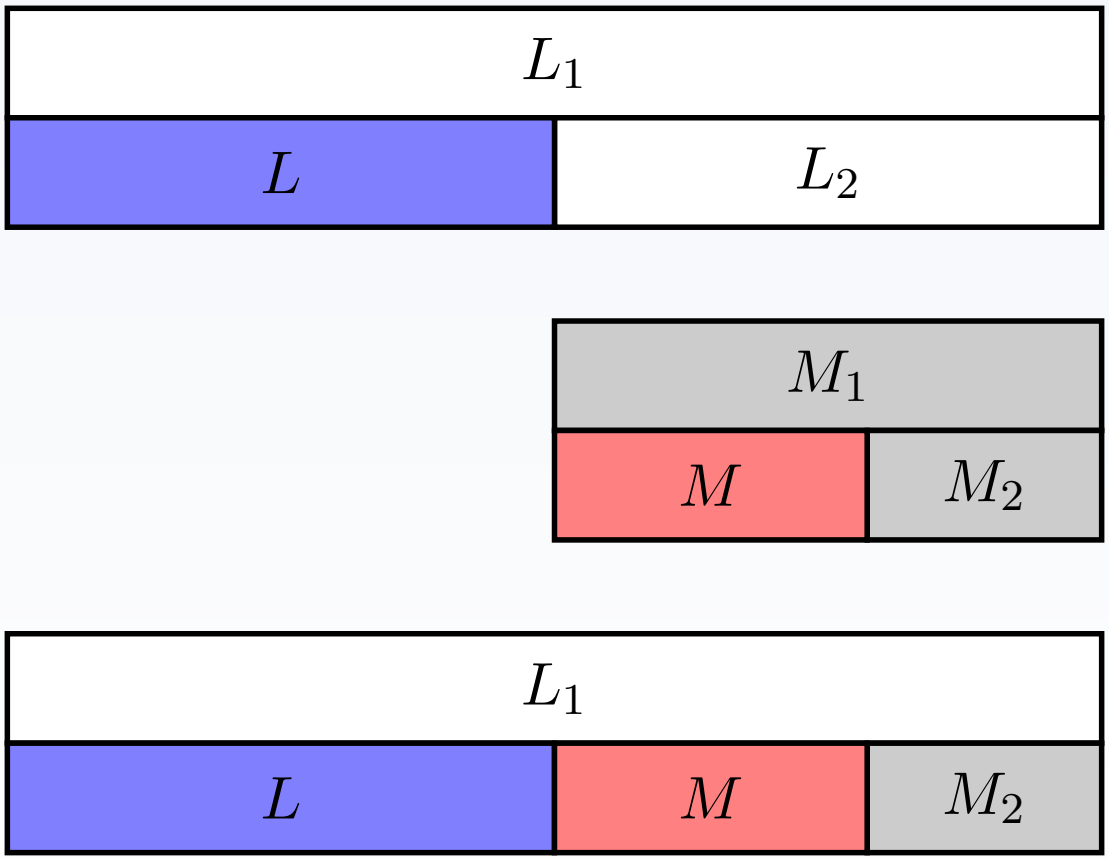
Differenzlisten

Darstellung der Liste L als Differenz $L_1 - L_2$



append mit Differenzlisten

$\texttt{append}(L_1 - L_2, M_1 - M_2, L_1 - M_2)$:- $L_2 = M_1$



Beachte: $L_2 = M_1$ muss gelten, sonst funktioniert es nicht

append mit Differenzlisten

In Prolog:

```
appendD(L1 - L2,M1 - M2,L1 - M2) :- L2 = M1.
```

Oder einfacher:

```
appendD(L1 - L2,L2 - M2,L1 - M2).
```

Anfrage
`append(Liste1 - Liste1Rest, Liste2 - Liste2, Ergebnis)`

- ist in konstanter Zeit beantwortbar, wenn `Liste1Rest` eine Variable ist.

```
?- appendD([1,2,3|Y]-Y,[4,5,6|Z]-Z,R).
Y = [4, 5, 6|Z],
R = [1, 2, 3, 4, 5, 6|Z]-Z
```

Der Cut-Operator

- Dient zum Abschneiden der Suche
- Operator: !

$$B \Leftarrow A_1, \ldots, A_m, !, A_{m+1}, \ldots, A_n$$

- Wenn Suche entlang $A_1, \ldots, A_m$ erfolgreich
- Backtracking von A_i mit $i \in \{m+1, \ldots, A_n\}$ überspringt $A_1, \ldots, A_m$
- Die entsprechenden Wege im SLD-Baum werden abgeschnitten

Der Cut-Operator: Beispiel

- Mitarbeiter des Wetteramts muss je nach Windstärke X
- eine der drei Mitteilungen $Y =$ „normal“, „windig“ oder „stürmisch“ weitergeben.

Regeln:

1. Wenn $X < 4$, dann $Y = normal$.
2. Wenn $4 \leq X$ und $X < 8$ dann $Y = windig$.
3. Wenn $8 \leq X$ dann $Y = stuermisch$.

Der Cut-Operator: Beispiel

- Mitarbeiter des Wetteramts muss je nach Windstärke X
- eine der drei Mitteilungen $Y =$ „normal“, „windig“ oder „stürmisch“ weitergeben.

Regeln:

1. Wenn $X < 4$, dann $Y = normal$.
2. Wenn $4 \leq X$ und $X < 8$ dann $Y = windig$.
3. Wenn $8 \leq X$ dann $Y = stuermisch$.

Prolog-Programm dazu:

```
mitteilung(X,normal) :- X < 4.
mitteilung(X,windig) :- 4 =< X, X < 8.
mitteilung(X,stuermisch) :- 8 =< X.
```

Der Cut-Operator: Beispiel (2)

```
?- mitteilung(3,Y), Y=windig.
```

Trace:

```
[trace] 3 ?- mitteilung(3,Y), Y=windig.
   Call: (7) mitteilung(3, _G2797) ? creep
   Call: (8) 3<4 ? creep
   Exit: (8) 3<4 ? creep
   Exit: (7) mitteilung(3, normal) ? creep
   Call: (7) normal=windig ? creep
   Fail: (7) normal=windig ? creep
   Redo: (7) mitteilung(3, _G2797) ? creep
   Call: (8) 4=<3 ? creep
   Fail: (8) 4=<3 ? creep
   Redo: (7) mitteilung(3, _G2797) ? creep
   Call: (8) 8=<3 ? creep
   Fail: (8) 8=<3 ? creep
   Fail: (7) mitteilung(3, _G2797) ? creep
false.
```

Betrachtung der 2. und 3. Programmklausel eigentlich unnötig!

Der Cut-Operator: Beispiel (3)

Mit Cut:

```
mitteilung(X,normal) :- X < 4,!.
mitteilung(X,windig) :- 4 =< X, X < 8,!.
mitteilung(X,stuermisch) :- 8 =< X.
```

Trace:

```
[trace] 4 ?-
|    mitteilung(3,Y), Y=windig.
   Call: (7) mitteilung(3, _G2803) ? creep
   Call: (8) 3<4 ? creep
   Exit: (8) 3<4 ? creep
   Exit: (7) mitteilung(3, normal) ? creep
   Call: (7) normal=windig ? creep
   Fail: (7) normal=windig ? creep
false.
```

Der Cut-Operator: Beispiel (4)

Mit Cut:

```
mitteilung(X,normal) :- X < 4,!.
mitteilung(X,windig) :- 4 =< X, X < 8,!.
mitteilung(X,stuermisch) :- 8 =< X.
```

Programm mit Cuts logisch äquivalent zum Programm ohne Cuts!
(Nur die operationale Semantik (Suche) hat sich verändert)

Der Cut-Operator: Beispiel (5)

Weitere Verbesserung des Programms:

```
mitteilung(X,normal) :- X < 4,!.
mitteilung(X,windig) :- X < 8,!.
mitteilung(X,stuermisch).
```

Beachte: Diese Implementierung nicht mit direktem Wert für Y aufrufen:

```
?- mitteilung(3,windig).
true.
?- mitteilung(3,Y), Y=windig.
false.
```

Der Cut-Operator: Beispiel (6)

```
mitteilung(X,normal) :- X < 4,!.
mitteilung(X,windig) :- X < 8,!.
mitteilung(X,stuermisch).
```

Weglassen der Cuts: **Andere Semantik!**

```
mitteilung(X,normal) :- X < 4.
mitteilung(X,windig) :- X < 8.
mitteilung(X,stuermisch).
```

```
?- mitteilung(3,Y), Y=windig.
true.
```

Fazit

- Cuts können die Effizienz steigern
- aber: können die Semantik verändern
- Vorsicht ist geboten!
- Besser: Cuts nur verwenden, wenn gleich zur Semantik ohne Cuts
- Insbesondere aufgrund der Wartbarkeit

Beispiele zu Cut (1)

Maximum berechnen

Ohne Cut:

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- Y < X.
```

Mit Cut:

```
max(X,Y,X) :- X >= Y,!.
max(X,Y,Y).
```

Achtung `?- max(3,1,1)` ergibt wahr!

Daher `max(3,1,Z)`, `Z=3` verwenden.

Beispiele zu Cut (2)

member

Ohne Cut:

```
member(X,[X|_]).
member(X,[_|Y]) :-  member(X,Y).
```

Mit Cut:

```
member(X,[X|_]) :-!.
member(X,[_|Y]) :-  member(X,Y).
```

Version mit Cut:

```
?- member(X,[1,2,3]).
X=1.
```

Version ohne Cut:

```
?- member(X,[1,2,3]).
X=1;
X=2;
X=3.
```

If-then-else

if-then-else: Eigentlich nicht darstellbar

Mit Cut:

```
ifthenelse(B,P,Q) :- B,!,P.
ifthenelse(B,P,Q) :- Q.
```

Nochmal: Semantikveränderung

```
p :- a,b.
p :- c.
```

```
p :- c.
p :- a,b.
```

Semantik beider Programme: $(a \wedge b \Rightarrow p) \wedge (c \Rightarrow p)$

```
p :- a,!,b.
p :- c.
```

```
p :- c.
p :- a,!,b.
```

$(a \wedge b \Rightarrow p) \wedge (\neg a \wedge c \Rightarrow p)$

$(c \Rightarrow p) \wedge (a \wedge b \Rightarrow p)$

Negation und not

- Beachte: Negation in Programmklauseln geht nicht
- $A \Leftarrow \neg B$ entspräche Klausel $\{A, B\}$
- Keine definite Klausel!

Aber: Es gibt `fail` in Prolog: Interpretation: Fehlschlagen der Suche

Beispiel

Maria mag alle Tiere außer Schlangen.

```
mag(maria,Y) :- Y=schlange,!,fail.
mag(maria,Y) :- tier(Y).
```

```
?- mag(maria,schlange).
false.

?- mag(maria,hund).
true.

?- mag(maria,katze).
true.
```

Closed World Assumption

Prolog verwendet die Closed World Assumption:

Fehlschlagen der Suche wird als Falsch interpretiert

Definition von `not`:

```
not(P) :- P,!,fail.
not(P) :- true.
```

Achtung: `not` entspricht nicht der logischen Negation!

Beispiel

Unser Beispiel kann daher auch programmiert werden als:

```
mag1(maria,Y) :- tier(Y), not(Y = schlange).
```

Closed-World-Assumption

```
rund(ball).
```

```
?- rund(ball).
true.

? rund(erde).
false

? not(rund(erde)).
true
```

Obwohl `not(rund(erde))` nicht herleitbar, liefert Prolog `true`!

Weitere Beispiele (1)

```
frau(maria).
...
mann(peter).
...
vater(peter,maria).
...
mutter(susanne,monika).
...
```

```
bruder(X,Y)      :- mann(X),vater(Z,X),vater(Z,Y),
                    not(X == Y), mutter(M,X),mutter(M,Y).
schwester(X,Y)   :- frau(X),vater(Z,X),vater(Z,Y),
                    not(X == Y),mutter(M,X),mutter(M,Y).
geschwister(X,Y) :- vater(Z,X),vater(Z,Y),
                    not(X == Y),mutter(M,X),mutter(M,Y).
```

Weitere Beispiele: Ungerichteter Graph

```
kante(a,b).
kante(a,c).
kante(c,d).
kante(d,b).
kante(b,e).
kante(f,g).

verbunden(X,Y) :- kante(X,Y).
verbunden(X,Y) :- kante(Y,X).
verbunden(X,Z) :- kante(X,Y),verbunden(Y,Z).
verbunden(X,Z) :- kante(Y,X),verbunden(Y,Z).

%terminierend:
verbunden2(X,Y) :- verb(X,Y,[]).
ungkante(X,Y) :-  kante(X,Y).
ungkante(X,Y) :-  kante(Y,X).
verb(X,Y,_) :- ungkante(X,Y).
verb(X,Z,L) :- ungkante(X,Y), not(member(Y,L)), verb(Y,Z,[Y|L]).
```

Vergleich: Theorie und Reale Implementierungen

Pures Prolog Definite Programme	nichtpures Prolog Cut, Negation Klauselreihenfolge fest	nichtpur, ohne occurs-check
SLD: ist korrekt	SLD: korrekt	SLD: i.a. nicht korrekt
vollständig	unvollständig	unvollständig

Prolog als Parser

- Prolog verwendet Tiefensuche mit Backtracking
- Das passt genau zu **rekursiv absteigenden Parsern**
- Eine wesentliche Anwendung von Prolog ist das Verarbeiten von Sprachen
- Insbesondere: Natural Language Processing

In Prolog gibt es sogenannte **Definite Clause Grammars** (DCG)

- Direkte Repräsentation für kontextfreie Grammatiken
- Aber: Erweiterte kontextfreie Grammatiken

Kontextfreie Grammatiken

CFGs bestehen aus Terminalen, Nichtterminalen, Produktionen

Beispiel:

$$\begin{aligned} S &\rightarrow ab \\ S &\rightarrow aSb \end{aligned}$$

erzeugt alle Worte der Form $a^n b^n$.

In Prolog als DCG:

```
s --> [a,b].
s --> [a], s, [b].
```

- statt $\rightarrow$ wird `-->` benutzt
- Terminale werden in Listenklammern geschrieben
- Folgen werden durch Kommata getrennt

DCGs

DCGs sind nur **syntaktischer Zucker**:

Sie werden in Prologklauseln übersetzt:

- Jedes Nichtterminal s wird zu einem zweistelligen Prädikat: $s(L_1, L_2)$
- Dabei ist (L_1, L_2) die Eingabeliste, dargestellt als **Differenzliste**

```
?- s([a,a,b,b],[]).
true.
?- s([a,b,b,a],[]).
false.
?- s([a,a,a,b,b,b],[]).
true.
?- s([a,a,a,b,b,b|X],X).
true.
?- s([a,a,a,b,b,b,c,d],[c,d]).
true.
```

Übersetzung: CFG → Klauseln

Produktion: `n --> n1,...,nm`

Annahme zunächst: Alle `ni` sind Nichtterminale

Übersetzung in eine Klausel:

```
n(Ein,Rest) :- n1(Ein,R1),n2(R1,R2),...,nm(Rm,Rest)
```

Beachte: Die Liste wird sequentiell auf die Nichtterminale aufgeteilt.

Übersetzung: CFG $\rightarrow$ Klauseln (2)

Übersetzung der Terminale: Prädikat `terminal`

```
terminal(Terminale,Eingabe,Rest) :-
        append(Terminale,Rest,Eingabe).
```

Übersetzung: Wenn ni gerade die Liste `[t1,...,tk]` von Terminalen ist:

Ersetze ni durch `terminal([t1,....,tk],R`$I-1$`,R`I`)`

Andere Variante: Direkt kodieren ohne `terminal`

Beispiel

```
s --> [a,b].
s --> [a], s, [b].
```

Beispiel

```
s(Ein,Rest) :- terminal([a,b],Ein,Rest).
s --> [a], s, [b].
```

Beispiel

```
s(Ein,Rest) :- terminal([a,b],Ein,Rest).
s(Ein,Rest) :- terminal([a],Ein,R1),s(R1,R2),terminal([b],R2,Rest).
```

Beispiel

```
s(Ein,Rest) :- terminal([a,b],Ein,Rest).
s(Ein,Rest) :- terminal([a],Ein,R1),s(R1,R2),terminal([b],R2,Rest).
```

Übersetzung ohne `terminal`:

```
s([a,b|Ein],Ein).
s([a|Ein],Rest) :-  s(Ein,[b|Rest]).
```

Parametrisierte CFGs

- DCGs können mehr (auch nicht kontextfreie) Sprachen darstellen, denn
- Nichtterminale können zusätzlich mit Attributen (Parametern) versehen werden:
  `s(X1,X2,...) --> ... s(X1',X2',...) ...`
- Diese werden mitunifiziert
- Jedes Auftreten des Nichtterminals muss gleich viele Parameter haben

Beispiel

$\{a^n b^n c^n d^n | n \in \mathbb{N}\}$ ist nicht kontextfrei, DCG dazu:

```
s             --> aa(X),bb(X),cc(X),dd(X).
aa(0)         --> [a].
aa(succ(X))   --> [a], aa(X).
bb(0)         --> [b].
bb(succ(X))   --> [b], bb(X).
cc(0)         --> [c].
cc(succ(X))   --> [c], cc(X).
dd(0)         --> [d].
dd(succ(X))   --> [d], dd(X).
```

- Der zusätzliche Parameter ist eine Peano-Zahl (Darstellung mit succ und 0)
- `s --> aa(X),bb(X),cc(X),dd(X).` sichert zu, dass gleich viele a's,b's,c's und d's erzeugt werden.

Beispiel (2)

Testen:

```
?- s([a,a,b,b,c,c,d,d],[]).
true .

?- s([a,a,a,b,b,c,c,d,d],[]).
false.

?- s(Eingabe,[]).
Eingabe = [a, b, c, d] ;
Eingabe = [a, a, b, b, c, c, d, d] ;
Eingabe = [a, a, a, b, b, b, c, c, c|...]
...
```

Beispiel (3)

Übersetzung in Prolog-Klauseln:

```
s             --> aa(X),bb(X),cc(X),dd(X).
aa(0)         --> [a].
aa(succ(X))   --> [a], aa(X).
bb(0)         --> [b].
bb(succ(X))   --> [b], bb(X).
cc(0)         --> [c].
cc(succ(X))   --> [c], cc(X).
dd(0)         --> [d].
dd(succ(X))   --> [d], dd(X).
```

Beispiel (3)

Übersetzung in Prolog-Klauseln:

```
s(Ein,Rest) :- aa(X,Ein,R1),bb(X,R1,R2),cc(X,R2,R3),dd(X,R3,Rest).
aa(0)       --> [a].
aa(succ(X)) --> [a], aa(X).
bb(0)       --> [b].
bb(succ(X)) --> [b], bb(X).
cc(0)       --> [c].
cc(succ(X)) --> [c], cc(X).
dd(0)       --> [d].
dd(succ(X)) --> [d], dd(X).
```

Beispiel (3)

Übersetzung in Prolog-Klauseln:

```
s(Ein,Rest) :- aa(X,Ein,R1),bb(X,R1,R2),cc(X,R2,R3),dd(X,R3,Rest).
aa(0,Ein,Rest) :- terminal([a],Ein,Rest).
aa(succ(X))  --> [a], aa(X).
bb(0)        --> [b].
bb(succ(X))  --> [b], bb(X).
cc(0)        --> [c].
cc(succ(X))  --> [c], cc(X).
dd(0)        --> [d].
dd(succ(X))  --> [d], dd(X).
```

Beispiel (3)

Übersetzung in Prolog-Klauseln:

```
s(Ein,Rest) :- aa(X,Ein,R1),bb(X,R1,R2),cc(X,R2,R3),dd(X,R3,Rest).
aa(0,Ein,Rest) :- terminal([a],Ein,Rest).
aa(succ(X),Ein,Rest) :- terminal([a],Ein,R1),  aa(X,R1,Rest).
bb(0)        --> [b].
bb(succ(X))  --> [b], bb(X).
cc(0)        --> [c].
cc(succ(X))  --> [c], cc(X).
dd(0)        --> [d].
dd(succ(X))  --> [d], dd(X).
```

Beispiel (3)

Übersetzung in Prolog-Klauseln:

```
s(Ein,Rest) :- aa(X,Ein,R1),bb(X,R1,R2),cc(X,R2,R3),dd(X,R3,Rest).
aa(0,Ein,Rest) :- terminal([a],Ein,Rest).
aa(succ(X),Ein,Rest) :- terminal([a],Ein,R1),  aa(X,R1,Rest).
bb(0,Ein,Rest) :- terminal([b],Ein,Rest).
bb(succ(X))  --> [b], bb(X).
cc(0)        --> [c].
cc(succ(X))  --> [c], cc(X).
dd(0)        --> [d].
dd(succ(X))  --> [d], dd(X).
```

Beispiel (3)

Übersetzung in Prolog-Klauseln:

```
s(Ein,Rest) :- aa(X,Ein,R1),bb(X,R1,R2),cc(X,R2,R3),dd(X,R3,Rest).
aa(0,Ein,Rest) :- terminal([a],Ein,Rest).
aa(succ(X),Ein,Rest) :- terminal([a],Ein,R1),  aa(X,R1,Rest).
bb(0,Ein,Rest) :- terminal([b],Ein,Rest).
bb(succ(X),Ein,Rest) :- terminal([b],Ein,R1),  bb(X,R1,Rest).
cc(0)        --> [c].
cc(succ(X))  --> [c], cc(X).
dd(0)        --> [d].
dd(succ(X))  --> [d], dd(X).
```

Beispiel (3)

Übersetzung in Prolog-Klauseln:

```
s(Ein,Rest) :- aa(X,Ein,R1),bb(X,R1,R2),cc(X,R2,R3),dd(X,R3,Rest).
aa(0,Ein,Rest) :- terminal([a],Ein,Rest).
aa(succ(X),Ein,Rest) :- terminal([a],Ein,R1),  aa(X,R1,Rest).
bb(0,Ein,Rest) :- terminal([b],Ein,Rest).
bb(succ(X),Ein,Rest) :- terminal([b],Ein,R1),  bb(X,R1,Rest).
cc(0,Ein,Rest) :- terminal([c],Ein,Rest).
cc(succ(X))  --> [c], cc(X).
dd(0)        --> [d].
dd(succ(X))  --> [d], dd(X).
```

Beispiel (3)

Übersetzung in Prolog-Klauseln:

```
s(Ein,Rest) :- aa(X,Ein,R1),bb(X,R1,R2),cc(X,R2,R3),dd(X,R3,Rest).
aa(0,Ein,Rest) :- terminal([a],Ein,Rest).
aa(succ(X),Ein,Rest) :- terminal([a],Ein,R1),  aa(X,R1,Rest).
bb(0,Ein,Rest) :- terminal([b],Ein,Rest).
bb(succ(X),Ein,Rest) :- terminal([b],Ein,R1),  bb(X,R1,Rest).
cc(0,Ein,Rest) :- terminal([c],Ein,Rest).
cc(succ(X),Ein,Rest) :- terminal([c],Ein,R1),  cc(X,R1,Rest).
dd(0)         --> [d].
dd(succ(X))   --> [d], dd(X).
```

Beispiel (3)

Übersetzung in Prolog-Klauseln:

```
s(Ein,Rest) :- aa(X,Ein,R1),bb(X,R1,R2),cc(X,R2,R3),dd(X,R3,Rest).
aa(0,Ein,Rest) :- terminal([a],Ein,Rest).
aa(succ(X),Ein,Rest) :- terminal([a],Ein,R1),  aa(X,R1,Rest).
bb(0,Ein,Rest) :- terminal([b],Ein,Rest).
bb(succ(X),Ein,Rest) :- terminal([b],Ein,R1),  bb(X,R1,Rest).
cc(0,Ein,Rest) :- terminal([c],Ein,Rest).
cc(succ(X),Ein,Rest) :- terminal([c],Ein,R1),  cc(X,R1,Rest).
dd(0,Ein,Rest) :- terminal([d],Ein,Rest).
dd(succ(X))  --> [d], dd(X).
```

Beispiel (3)

Übersetzung in Prolog-Klauseln:

```
s(Ein,Rest) :- aa(X,Ein,R1),bb(X,R1,R2),cc(X,R2,R3),dd(X,R3,Rest).
aa(0,Ein,Rest) :- terminal([a],Ein,Rest).
aa(succ(X),Ein,Rest) :- terminal([a],Ein,R1),  aa(X,R1,Rest).
bb(0,Ein,Rest) :- terminal([b],Ein,Rest).
bb(succ(X),Ein,Rest) :- terminal([b],Ein,R1),  bb(X,R1,Rest).
cc(0,Ein,Rest) :- terminal([c],Ein,Rest).
cc(succ(X),Ein,Rest) :- terminal([c],Ein,R1),  cc(X,R1,Rest).
dd(0,Ein,Rest) :- terminal([d],Ein,Rest).
dd(succ(X),Ein,Rest) :- terminal([d],Ein,R1),  dd(X,R1,Rest).
```

Beispiel (4)

Übersetzung in Prolog-Klauseln ohne `terminal`:

```
s(Ein,Rest) :- aa(X,Ein,R1),bb(X,R1,R2),cc(X,R2,R3),dd(X,R3,Rest).
aa(0,[a|Ein],Ein).
aa(succ(X),[a|Ein],Rest) :- aa(X,Ein,Rest).
bb(0,[b|Ein],Ein).
bb(succ(X),[b|Ein],Rest) :- bb(X,Ein,Rest).
cc(0,[c|Ein],Ein).
cc(succ(X),[c|Ein],Rest) :- cc(X,Ein,Rest).
dd(0,[d|Ein],Ein).
dd(succ(X),[d|Ein],Rest) :- dd(X,Ein,Rest).
```

Natürliche Sprachverarbeitung

- Grammatik der Sprache als DCG
- Attribute sorgen dafür, dass Geschlecht, Fall, Zeit usw. übereinstimmen
- Wir betrachten Englisch, da z.B. Deutsch komplizierter ist.

Syntaktische Kategorien

Det	Determiner (Artikel)	(the, a, some)
N	Noun (Nomen)	(table, computer, John)
V	verb	(writes, eats, having)
ADJ	adjectives	(big, fast)
ADV	adverbs	(very, slowly, yesterday)
AUX	auxiliaries (Hilfsverben)	(is, do, has will)
CON	conjunctions	(and, or)
PREP	prepositions	(to, on, with)
PRON	Pronouns (Pronomen)	(he, who, which)

Subkategorien:

S	Sentence	(Satz)
PN	proper noun	(“a .... “ ist verboten)
IV	intransitive verb	hat kein Objekt
TV	transitives verb	hat Objekt.

Komponenten eines Satzes

Phrasen, Sätze:

S	(Sentence) Satz	
NP	Nounphrase Nominalphrase	das dicke Buch
VP	Verbphrase	schreibe ein Buch
PP	Präpositionalphrase	mit einem Fernglas
ADJP	adjective phrase	größer als man erwartet
ADVP	adverbial phrase	(gestern abend)
OptRel	relative clause, optional	

Phrasenstrukturregeln (vereinfacht)

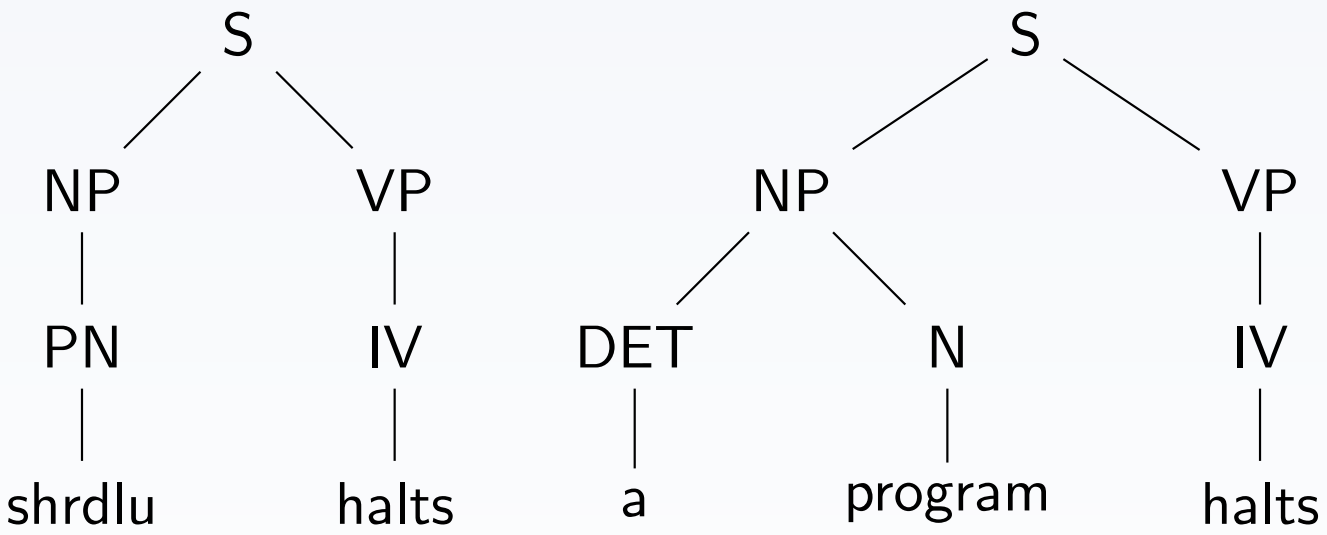
Als Kontextfreie Grammatik:

S	$\rightarrow$	NP VP
NP	$\rightarrow$	Det N
NP	$\rightarrow$	Det N OptRel
NP	$\rightarrow$	PN
Optrel	$\rightarrow$	that VP
VP	$\rightarrow$	TV NP
VP	$\rightarrow$	IV

Lexikoneinträge:

PN	$\rightarrow$	terry
PN	$\rightarrow$	shrdlu
Det	$\rightarrow$	a
N	$\rightarrow$	program
IV	$\rightarrow$	halts
TV	$\rightarrow$	writes

Beispielherleitungsbäume



Grammatik in Prolog

```
istsatz(Ein) :-  s(Ein,[]).

s       --> np, vp.
np      --> pn.
np      --> det, n.
np      --> det, n, optrel.
optrel --> [that], vp.
vp      --> tv, np.
vp      --> iv.
% Lexikon:
pn      --> [terry].
pn      --> [shrdlu].
det     --> [a].
n       --> [program].
iv      --> [halts].
tv      --> [writes].
```

Beispielaufrufe

```
?- istsatz([a,program,halts]).
true .
?- istsatz([shrdlu,halts]).
true .
?- istsatz([terry,writes,a,program,that,halts]).
true .
?- istsatz([terry,halts,a,program]).
false.
?- istsatz([terry,writes,a,shrdlu]).
false.
?- istsatz([terry,writes,a,program]).
true .
?- istsatz([terry,writes,shrdlu]).
true .
?- istsatz(X).
X = [terry, writes, terry] ;
X = [terry, writes, shrdlu] ;
X = [terry, writes, a, program] ;
X = [terry, writes, a, program, that, writes, terry] ;
X = [terry, writes, a, program, that, writes, shrdlu] ;
X = [terry, writes, a, program, that, writes, a, program] ;
...
```

Übersetzung in Prologklauseln

```
istsatz(Ein) :-  s(Ein,[]).

s(Ein,Rest)                 :- np(Ein,Rest1), vp(Rest1,Rest).
np(Ein,Rest)                :- pn(Ein,Rest).
np(Ein,Rest)                :- det(Ein,Rest1), n(Rest1,Rest).
np(Ein,Rest)                :- det(Ein,Rest1),n(Rest1,Rest2),
                               optrel(Rest2,Rest).
optrel([that|Ein],Rest)     :- vp(Ein,Rest).
vp(Ein,Rest)                :- tv(Ein,Rest1), np(Rest1,Rest).
vp(Ein,Rest)                :- iv(Ein,Rest).
% Lexikon:
pn([terry|Ein],Ein).
pn([shrdlu|Ein],Ein).
det([a|Ein],Ein).
n([program|Ein],Ein).
iv([halts|Ein],Ein).
tv([writes|Ein],Ein).
```

Verwendung von DCGs

- Die so definierte Grammatik erlaubt viele ungültige Sätze
- Z.B. sollte der Numerus von NP und VP übereinstimmen: D.h. singular oder plural: „sie gehen“, „er geht“ aber nicht „er gehen“.
- Lösung: Kodierung durch Attribute in der DCG

Grammatik mit Attributen

```
s               -->  np(Number), vp(Number).
np(Number)      -->  pn(Number).
vp(Number)      -->  tv(Number), np(_).
vp(Number)      -->  iv(Number).
pn(singular)    -->  [shrdlu].
pn(plural)      -->  [they].
iv(singular)    -->  [halts].
iv(plural)      -->  [halt].
tv(singular)    -->  [writes].
tv(plural)      -->  [write].
```

Grammatik mit Attributen: Übersetzung

```
s(P0,P)              :- np(Number,P0,P1), vp(Number,P1,P).
np(Number,P0, P)     :- pn(Number,P0,P).
vp(Number,P0, P)     :- tv(Number, P0, P1), np(_, P1, P).
vp(Number,P0, P)     :- iv(Number, P0, P).
pn(singular,P0,P)    :- terminal([shrdlu], P0, P).
pn(plural, P0, P)    :- terminal([they],P0, P).
iv(singular, P0, P):- terminal([halts], P0, P).
iv(plural, P0, P)    :- terminal([halt], P0, P).
tv(singular,P0,P)    :- terminal([writes],P0,P).
tv(plural,P0,P)      :- terminal([write],P0,P).
```

Verarbeitung

Die Anfrage `s([shrdlu, halts],[])` hat folgende Verarbeitung:

```
s([shrdlu, halts],[])
  np(N,[shrdlu, halts],P1), vp(N,P1,[])
  pn(N,[shrdlu, halts],P1), iv(N,P1,[])
  terminal(shrdlu,[shrdlu, halts],P1),  iv(singular,P1,[])
                %%%   (N = singular)
  terminal(shrdlu,[shrdlu, halts],P1),  terminal([halts],P1,[])
                %%%  (Unifikation mit terminal ergibt P1 = [halts])
  terminal([halts],[halts],[]).
Ergibt ‘‘yes‘‘.
```

Features

- Es gibt neben Numerus noch weitere Attribute
- z.B. Person (erste, zweite dritte): „ich bin“ nicht „ich bist“
- Geschlecht: „das Haus“ nicht „die Haus“
- diese Attribute werden in der Computerlinguistik auch Features genannt

DCG mit Prolog-Code

- Produktionen dürfen zusätzlich Prologcode enthalten
- Dieser muss in geschweifte Klammern gesetzt werden

DCG mit Prolog-Code

- Produktionen dürfen zusätzlich Prologcode enthalten
- Dieser muss in geschweifte Klammern gesetzt werden

Beispiel:

```
genA(0) --> [].
genA(A) --> {A > 0, A1 is A - 1}, [a], genA(A1).
```

Aufruf z.B.

```
?- genA(2,X,[]).
X = [a, a] .

?- genA(4,X,[]).
X = [a, a, a, a] .

?- genA(6,X,[]).
X = [a, a, a, a, a, a] .
```

Kleine DCG des Deutschen

```
s                          --> np(Number,Sex,Person), vp(Number,Sex,Person).

np(Number,Sex,Person)      --> pn(Number,Sex,Person,nom).

vp(Number,Sex,Person)      --> tv(Number,Sex,Person,Semtv),
                               npakk(akk,Semnp),
                               {intersect(Semtv,Semnp,Semschnitt), not (Semschnitt  = [])}.
vp(Number,Sex,Person)      --> iv(Number,Sex,Person).

pn(singular,no,1,nom)      --> [ich].
pn(singular,no,2,nom)      --> [du].
pn(singular,male,3,nom)    --> [er].
pn(singular,female,3,nom)  --> [sie].
pn(singular,neutrum,3,nom) --> [es].
pn(plural,no,1,nom)        --> [wir].
pn(plural,no,2,nom)        --> [ihr].
pn(plural,mfn,3,nom)       --> [sie].

npakk(Fall,Sem)            --> det(Number,Sex,3,Fall),nom(Number,Sex,3,Fall,Sem).

iv(singular,male,3)        --> [geht].
iv(singular,female,3)      --> [geht].
iv(singular,neutrum,3)     --> [geht].
iv(singular,no,1)          --> [gehe].
iv(singular,no,2)          --> [gehst].
iv(plural,mfn,3)           --> [gehen].
iv(plural,no,1)            --> [gehen].
iv(plural,no,2)            --> [geht].
```

Kleine DCG des Deutschen (2)

```
tv(singular,male,3,[transport])      -->  [f\"ahrt].
tv(singular,female,3,[transport])    -->  [f\"ahrt].
tv(singular,neutrum,3,[transport])   -->  [f\"ahrt].
tv(singular,no,1,[transport])        -->  [fahre].
tv(singular,no,2,[transport])        -->  [f\"ahrst].
tv(plural,mfn,3,[transport])         -->  [fahren].
tv(plural,no,1,[transport])          -->  [fahren].
tv(plural,no,2,[transport])          -->  [fahrt].
tv(singular,male,3,[information])    -->  [liest].
tv(singular,female,3,[information])  -->  [liest].
tv(singular,neutrum,3,[information]) -->  [liest].
tv(singular,no,1,[information])      -->  [lese].
tv(singular,no,2,[information])      -->  [liest].
tv(plural,mfn,3,[information])       -->  [lesen].
tv(plural,no,1,[information])        -->  [lesen].
tv(plural,no,2,[information])        -->  [lest].
det(singular,neutrum,3,akk) --> [ein].
det(singular,neutrum,3,akk) --> [kein].
det(plural,neutrum,3,akk)   --> [zwei].
det(singular,male,3,akk)    --> [einen].
det(singular,male,3,akk)    --> [keinen].
det(plural,male,3,akk)      --> [zwei].
det(singular,female,3,akk)  --> [eine].
det(singular,female,3,akk)  --> [keine].
det(plural,female,3,akk)    --> [zwei].
nom(singular,neutrum,3,akk,[transport])  --> [auto].
nom(plural,neutrum,3,akk,[transport])    --> [autos].
nom(singular,male,3,akk,[transport])     --> [bus].
nom(plural,male,3,akk,[transport])       --> [busse].
nom(singular,female,3,akk,[information]) --> [zeitung].
nom(plural,female,3,akk,[information])   --> [zeitungen].
```

Lexikon

- Das Lexikon ist die zentrale Struktur der linguistischen Verarbeitung.
- Kodiere viel ins Lexikon
  Z.B. „laufen“: Passiv-Verbot: („ich werde gelaufen“)
- Erleichtert die Konstruktion der eigentlichen Grammatik.

```
take: verb
 verbtype = transitive
 subject: role   = agent, semfeat= human
 object:  role   = instrument, semfeat=vehicle
 prep-obj: prep  = to, role=goal
 prep-obj: prep  = from, role=source, ...
```

Berechnung von Parse-Bäumen

Statt

```
s             -->  np(Number), vp(Number).
np(Number)    -->  pn(Number).
vp(Number)    -->  tv(Number), np(_).
vp(Number)    -->  iv(Number).
pn(singular)  -->  [shrdlu].
pn(plural)    -->  [they].
iv(singular)  -->  [halts].
iv(plural)    -->  [halt].
tv(singular)  -->  [writes].
tv(plural)    -->  [write].
```

erweitere um Argumente, die den Syntaxbaum darstellen

```
s(sentence(TR_np,TR_vp))            -->  np(TR_np,Number), vp(TR_vp,Number).
np(nounphrase(TR_pn),Number)        -->  pn(TR_pn,Number).
vp(verbphrase(TR_tv,TR_np),Number) -->  tv(TR_tv,Number), np(TR_np,_).
vp(verbphrase(TR_iv),Number)        -->  iv(TR_iv,Number).
pn(propernoun(shrdlu),singular)     -->  [shrdlu].
pn(propernoun(they),plural)         -->  [they].
iv(intransverb(halts),singular)     -->  [halts].
iv(intransverb(halt),plural)        -->  [halt].
tv(transverb(writes),singular)      -->  [writes].
tv(transverb(write),plural)         -->  [write].
```

Fazit DCGs

- DCGs ohne Attribute und Prologcode sind CFGs
- Attribute erweitern die Mächtigkeit
- Formale Semantik: Durch die Übersetzung definierte Hornklauseln mit SLD-Resolution