

# PROLOG Einführung

1. Grundlagen
2. Syntax
3. Matching (Unifikation)
4. Semantik
5. Listen
6. Erweiterungen (Builtins)
7. Metalogische Konstrukte

# Grundlagen

- Repräsentation von Wissen
- Regeln
- Anfragen

# Repräsentation von Wissen

“Steffen ist Vater von Paul”

vater(steffen, paul).

“Paul ist Vater von Maria”

vater(paul, maria).

⇒ Repräsentation von Wissen (Relationen) als Fakten in  
einer PROLOG-Datenbasis.

## Anfragen (Goals)

“Ist Steffen der Vater von Paul?”

?– vater(steffen,paul).

“Wer ist Steffens Kind?”

?– vater(steffen,Kind).

Hinweis: Kind ist Variable,

PROLOG-**Variablen** fangen mit Großbuchstaben oder \_ an.

Konkrete Objekte (steffen,paul) heißen **Atome**.

# Konjunktive Anfragen

“Ist Steffen der Vater von Paul und Maria?”

?– vater(steffen,paul),vater(steffen,maria).

“Haben Paul und Maria denselben Vater?”

?– vater(X,paul),vater(X,maria).

Anfragen bestehen aus Teilanfragen (Subgoals).

# Familienbeziehungen

maennlich(paul). maennlich(fritz).

maennlich(steffen). maennlich(robert).

weiblich(karin). weiblich(lisa).

weiblich(maria). weiblich(sina).

vater(steffen,paul). vater(fritz,karin).

vater(steffen,lisa). vater(paul,maria).

mutter(karin,maria).

mutter(sina,paul).

PROLOG-Programme bestehen aus **Klauseln (clauses)**

# Repräsentation von Wissen

## Problem

Was passiert bei Einführung von grossvater?

“Steffen ist Großvater von Maria”

grossvater(steffen, maria).

⇒ Bei Hinzufügen von Fakten der vater-Relation  
(z.B. vater(paul, lena).) müssen die  
Großväter ergänzt werden

⇒ Idee: Großväter werden bei Bedarf “berechnet”

# Einführung von Regeln

$\text{grossvater}(X,Y) :- \text{vater}(X,Z), \text{vater}(Z,Y).$

$?- \text{grossvater}(X,\text{paul}).$

Bei Hinzufügen von Fakten zur vater-Relation bleibt die grossvater-Regel unverändert!

# Familienbeziehungen

elternteil(E,Kind) :- vater(E,Kind).

elternteil(E,Kind) :- mutter(E,Kind). *% oder*

vorfahre(X,Y) :- elternteil(X,Y).

vorfahre(X,Y) :- elternteil(X,Z), vorfahre(Z,Y).

bruder(X,Y) :- maennlich(X), elternteil(E,X),  
elternteil(E,Y), X \== Y.

Variablen gelten immer nur bis zum Punkt.

# PROLOG-Session

ECLiPSe Constraint Logic Programming System [sepia] Version 4.0,

Copyright ECRC GmbH and

ICL/IC-Parc, Fri Jul 17 15:48 1998

[eclipse 1]: [familie].

familie.pl compiled traceable 236 bytes in 0.00 seconds

yes.

[eclipse 2]: vater(X,Y).

X = steffen

Y = paul        More? (;)

X = paul

Y = maria

yes.

[eclipse 3]: vater(hans,X).

no (more) solution.

[eclipse 4]: halt.

bye

Program:

vater(steffen, paul).

vater(paul, maria).

# Wie kommt man an ein PROLOG-System?

<http://www.cs.uni-potsdam.de/wv/prolog/>

PROLOG Systeme am Institut

Link zur SWI-PROLOG Homepage

# Grund-PROLOG

- Syntax
- Matching
- Deklarativ und Prozedural Semantik
- Listen

# Syntax: **Atome** und **Konstanten**

- Folgen von alphanumerischen Zeichen beginnend mit kleinem Buchstaben (abxz)
- Folgen von Zeichen eingeschlossen in ' und ' ('\_u76')
- Sonderzeichen [], :-, etc.

sind **Atome**.

**Konstanten** sind **Atome** und **Zahlen**.

# Syntax: Variablen

- Variablen fangen mit Großbuchstaben an (X, List)
- **Anonyme** Variablen fangen mit \_ an (\_member,-)
- Anonyme Variablen werden von PROLOG nicht ausgegeben
- Der Gültigkeitsbereich (lexical scope) einer Variable ist immer eine Klausel, d.h....
- ...wenn X in zwei Klauseln vorkommt handelt es sich um zwei **verschiedene** Variablen!!!

# Syntax: Terme

- **Konstanten**
- **Variablen**
- **Zusammengesetzte Terme** ( $f(t_1, \dots, t_n)$ )

wobei  $t_i$  Term und  
 $f/n$   $n$ -stelliger Funktor

sind **Terme**.

**Beachte**

$$f(X, Y): f/2 \quad f(X): f/1 \quad f/1 \neq f/2$$

# Syntax: Horn Klauseln

- Fakten  
(weiblich(maria).)
- Regeln  
(vorfahre(X,Y) :- elternteil(X,Z), vorfahre(Z,Y).)
- Fakten und Regeln heißen **definite Klauseln**
- Ziele oder Anfragen  
(:- vater(paul, maria).)

**Logisches Programm** ist Folge von definiten Klauseln

# Matching

Zwei gegebene Terme **matchen** (passen zueinander) wenn

- sie identisch sind, oder
- die Variablen in beiden Termen so mit Objekten (Variablen, Atomen, anderen Termen) instanziiert werden können, daß nach der Ersetzung der Variablen durch diese Objekte identische Terme entstehen.

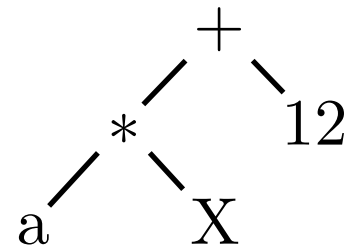
PROLOG-Terme werden auch als Strukturen bezeichnet und sind nicht mit Termen in anderen Sprachen (z.B. Aussagenlogik) zu verwechseln.

# Matching

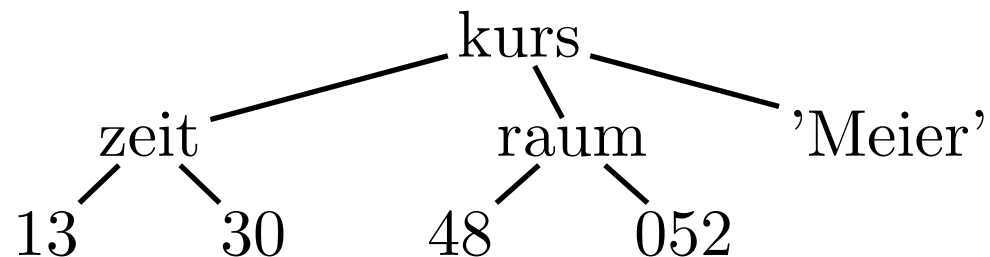
Term1	Term2	matchen
zeit(13,30)	zeit(13,X)	Ja
zeit(14,30)	zeit(13,X)	Nein
zeit(13,30)	zeit(13,30,05)	Nein
zeit(13,X,05)	zeit(Y,Z,05)	Ja
p(r(X,b),q(c,d))	p(r(a,b),Y)	Ja

# Terme als Bäume

$$(a * X) + 12$$



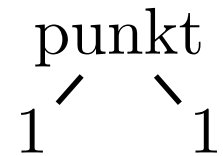
`kurs(zeit(13,30),raum(48,052),'Meier')`



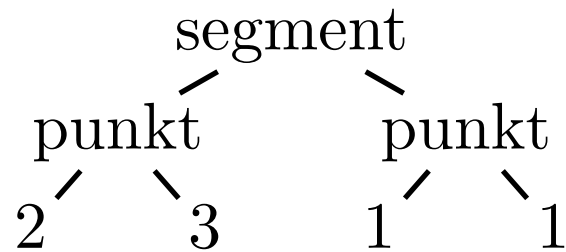
Die Wurzel des Baumes heißt **Funktor** und die Nachfolger der Wurzel heißen **Komponenten**.

# Beispiel

punkt(1,1)

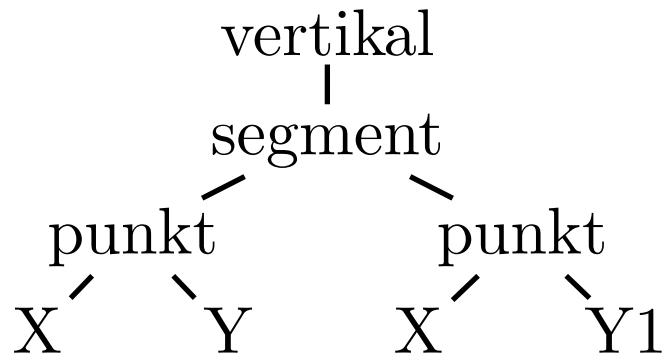


segment(punkt(2,3),punkt(1,1))



# Beispiel

vertikal( segment(punkt(X,Y),punkt(X,Y1))).



horizontal(segment(punkt(X,Y),punkt(X1,Y))). (analog)

# Beispiel

## Program

```
vertikal( segment(punkt(X,Y),punkt(X,Y1))).
```

```
horizontal(segment(punkt(X,Y),punkt(X1,Y))).
```

-----

```
[eclipse]: vertikal(segment(punkt(1,1),punkt(1,2))).
```

```
yes.
```

```
[eclipse]: vertikal(segment(punkt(1,1),punkt(2,Y))).
```

```
no (more) solution.
```

```
[eclipse]: horizontal(segment(punkt(1,1),punkt(2,Y))).
```

```
Y = 1
```

```
yes.
```

```
[eclipse]: vertikal(segment(punkt(2,3),P)).
```

```
P = punkt(2, Y1)
```

```
yes.
```

```
[eclipse]: vertikal(S),horizontal(S).
```

```
S = segment(punkt(X, Y), punkt(X, Y))
```

```
yes.
```

## Matching: Zusammenfassung

- Falls  $S$  und  $T$  Konstanten sind, matchen sie nur wenn sie identisch sind.
- Falls  $S$  Variable (beliebig) und  $T$  beliebig (Variable) ist, matchen  $S$  und  $T$ .  $S$  ( $T$ ) wird mit  $T$  ( $S$ ) instanziiert.
- Falls  $S$  und  $T$  Terme sind, matchen sie nur wenn
  - $S$  und  $T$  denselben Hauptfunktorkopf haben und
  - alle ihre korrespondierenden Komponenten matchen.

Die Instanziierung wird durch das matching der Komponenten bestimmt.

# Deklarative und prozedurale Semantik

- deklarative Semantik

Welche Anfragen lassen sich beantworten?

- prozedurale Semantik

Wie werden die Antworten gefunden?

# Deklarative und prozedurale Semantik

$P :- Q, R.$  wobei  $P, Q$  und  $R$  Terme sind.

- **deklarativ**

$P$  ist wahr wenn  $Q$  und  $R$  wahr sind oder

Aus  $Q$  und  $R$  folgt  $P$ .

- **prozedural**

Um Goal  $P$  zu erfüllen, erfülle *zuerst*  $Q$  und *dann*  $R$  oder

Um Problem  $P$  zu lösen, löse *zuerst*  $Q$  und *dann*  $R$ .

Es gibt andere prozedurale Semantiken (z.B. erst  $R$  dann  $Q$ ).

# Instanz und Variante

Bei einer **Instanz** einer Klausel  $C$  ist jede Variable durch Term ersetzt.

Bei einer **Variante** einer Klausel  $C$  ist jede Variable durch eine andere Variable ersetzt.

**Beispiel:**  $p(X) :- Q(X,Y).$

**Variante:**  $p(X1) :- Q(X1,Y1).$

**Instanz:**  $p(hans) :- Q(hans,vater(Y,hans)).$

# Deklarative Semantik

Ein Goal  $G$  ist **bzgl. eines Programms** wahr gdw

- es eine Klausel  $C$  im Programm gibt, so daß
- es eine Instanz  $I$  von  $C$  gibt, so daß gilt:
  1. der Head von  $I$  ist identisch mit  $G$ , und
  2. alle Goals im Body von  $I$  sind wahr.

Eine Liste von Goals ist wahr gdw jedes Goal der List bzgl. **derselben** Variableninstanziierung wahr ist.

# Prozedurale Semantik

Wie beantwortet PROLOG Anfragen?

- PROLOG versucht alle Teilanfragen zu erfüllen, d.h. zu zeigen, daß die Anfrage logisch aus den Fakten und Regeln folgen.
- Kommen Variablen in der Anfrage vor, versucht PROLOG diese mit Objekten zu instanziiieren, und dann die instanziierte Anfrage zu erfüllen.
- Ist eine Anfrage nicht aus der Datenbasis herleitbar, kann PROLOG sie nicht erfüllen.

# Prozedurale Semantik

**Execute**(Goal-Liste  $GL = G_1, G_2, \dots, G_m$ ):

- Falls  $GL$  leer return Yes, falls  $GL$  nicht leer weiter mit **Scan**.
- **Scan**: Gehe von oben nach unten durch das Programm, bis eine Klausel  $C$  gefunden ist, so daß  $\text{head}(C)$  und  $G_1$  matchen.

Falls es keine solche gibt return No,  
andernfalls hat  $C$  die Form

$$C = H : - B_1, \dots, B_n.$$

Sei  $C' = H' : - B_1', \dots, B_n'$

Variante von  $C$ , so daß  $C'$  und  $GL$  keine gemeinsamen Variablen haben.

Matche  $G_1$  und  $H'$  mit Variableninstanziierung  $I$ .

Setzte

$$GL' = B_1', \dots, B_n', G_2, \dots, G_m.$$

Ersetzte die Variablen in  $GL'$  gemäß  $I$ . Das ergibt eine neue Goal-Liste

$$GL'' = B_1'', \dots, B_n'', G_2', \dots, G_m'.$$

- Führe **Execute**( $GL''$ ) rekursiv aus. Falls neue Goal-Liste erfolgreich return Yes,  
andernfalls verwerfe neue Goal-Liste und gehe zurück zu **Scan**. Setzte **Scan** mit der Klausel direkt nach  $C$  fort.

# Beispiel

vater(steffen, paul).

vater(paul, maria).

mutter(maria, lisa).

(E1) elternteil(E,K) :– vater(E,K).

(E2) elternteil(E,K) :– mutter(E,K). % oder

(V1) vorfahre(X,Y) :– elternteil(X,Y).

(V2) vorfahre(X,Y) :– elternteil(X,Z), vorfahre(Z,Y).

# Beispiel

? :- vorfahre(steffen, maria).

```
0 CALL    vorfahre(steffen, maria)    [X = steffen, Y = maria] (V1)
1 CALL    elternteil(steffen, maria)  [E = steffen, K = maria] (E1)
2 CALL    vater(steffen, maria)
2 FAIL    vater(steffen, maria)       [backtracking]
1 NEXT    elternteil(steffen, maria)  [E = steffen, K = maria] (E2)
2 CALL    mutter(steffen, maria)
2 FAIL    mutter(steffen, maria)     [backtracking]
1 FAIL    elternteil(steffen, maria)  [backtracking]
0 NEXT    vorfahre(steffen, maria)    [X = steffen, Y = maria] (V2)
1 CALL    elternteil(steffen, Z)      [E = steffen, Z = ??]   (E1)
2 CALL    vater(steffen, Z)          [Z = paul]
2 EXIT    vater(steffen, paul)
1 *EXIT   elternteil(steffen, paul)
1 CALL    vorfahre(paul, maria)       [X = paul, Y = maria]   (V1)
2 CALL    elternteil(paul, maria)     [E= paul, Kind=maria]  (E1)
```

```
3 CALL    vater(paul, maria)
3 EXIT    vater(paul, maria)
2 *EXIT   elternteil(paul, maria)
1 *EXIT   vorfahre(paul, maria)
0 *EXIT   vorfahre(steffen, maria)
```

trace und notrace(nodebug) schaltet das Tracing ein bzw aus!!!

# Deklarative und prozedurale Semantik

Deklarative und prozedurale Semantik sind **nicht** identisch!

**Beispiel:**

$p \text{ :- } p.$

# Listen

## Listen in Termnotation

- leere Liste  $[]$
- (traditioneller) Funktor  $./2$
- $.(Kopf, Rest)$
- Listenende ist  $[]$

## Beispiele

$.(a, [])$

$.(1, X)$

$.(1, .(2, .(3, [])))$

$.(.(a, []), .(. (b, []), []))$

# Listen

[Kopf|Rest] statt  $.(Kopf,Rest)$

## Beispiele

$.(a,[])$

[a]

$.(1,X)$

[1|X]

$.(1,.(2,.(3,[])))$

[1,2,3]

$.(.(a,[]),.(.(b,[]),[]))$

[[a]||[b]] = [[a],[b]]

# Operationen auf Listen

## Durchlaufe eine Liste

liste([]).

liste([- | Rest]) :- liste(Rest).

## Element einer Liste

member(Ele,[Ele | \_]).

member(Ele,[\_ | Rest]) :- member(Ele,Rest).

# Operationen auf Listen

**Konkatenation** von Listen L1 und L2 zu L3

$\text{app}([],L,L).$

$\text{app}([E | L1],L2,[E | L3]) :- \text{app}(L1,L2,L3).$

# Erweiterungen des Grund-PROLOG

- Operatoren
- Arithmetik
- Klassifikation von Termen
- Input/Output
- Analyse und Synthese von Termen

# Operatoren

op(Präzedenz, Typ, OperatorName).

fx	Präfix	nicht assoziativ
fy	Präfix	rechtsassoziativ
xf	Postfix	nicht assoziativ
yf	Postfix	linksassoziativ
xfx	Infix	nicht assoziativ
yfx	Infix	linksassoziativ
xfy	Infix	rechtsassoziativ

# Operatoren

## Beispiel

?- op(400,xfx,ist\_vater\_von).

?- steffen ist\_vater\_von paul.

# Arithmetik

$X$  is *Expression*

- is unifiziert  $X$  und *Expression*
- *Expression* muß voll instanziiert sein!
- *Expression* wird dabei ausgewertet (im Gegensatz zu =)

# Arithmetik

+	Addition	-	Subtraktion
*	Multiplikation	/	Division
<	kleiner	>	größer
=<	kleiner gleich	>=	größer gleich
==	gleich	= / =	ungleich

## Beispiele

X is  $7 * 3$ .

$3 * 4$  is  $4 * 3$ .

12 is  $X + 4$ .

# Klassifikation von Termen

<code>number(X).</code>	Ist X eine Zahl?
<code>integer(X).</code>	Ist X eine ganze Zahl?
<code>float(X)</code>	Ist X eine Gleitkommazahl?
<code>atom(X)</code>	Ist X ein Atom?
<code>atomic(X)</code>	Ist X ein Atom oder eine Zahl?
<code>var(X)</code>	Ist X eine uninstanzierte Variable?
<code>nonvar(X)</code>	Ist X keine Variable?
<code>T1 == T2</code>	Sind T1 und T2 identisch?
<code>T1 \== T2</code>	Sind T1 und T2 nicht identisch?
<code>T1 = T2</code>	Sind T1 und T2 unifizierbar?

# Input/Output

put(X).	schreibe ASCII Zeichen
get(X).	lese druckbares ASCII Zeichen und unifiziere es mit X
get0(X).	lese ASCII Zeichen und unifiziere es mit X
tab(N).	erzeugt N Leerzeichen
nl.	Zeilenvorschub
write(X).	schreibt Term X
read(X).	lese nächsten Term und unifiziere mit X

put, get, get0, write und read gelingen nur einmal;  
sie werden beim backtracking übersprungen

Ähnliche Prädikate gibt es für den Zugriff auf Dateien.

# Analyse und Synthese von Termen

`functor(?Term,?FunctorName,?Arity).`

`arg(+N,+Term,-Argument).`

`?Term =.. ?List. (entspricht =..(Term,List))`

# Analyse und Synthese von Termen

## Beispiele

$\text{functor}(f(1,2),f,2).$

yes.

$\text{functor}(f(1,2),F,A).$

$F=f, A=2$

$\text{functor}(T,f,3).$

$T=f(-g50, -g52, -g54)$

$\text{functor}(T,..,2).$

$T=[-g48 \mid -g50]$

$\text{functor}(f(1,2),f,3).$

no (more) solution.

$\text{arg}(2,..(a,b,c),b).$

yes.

$\text{arg}(2,f(a,f(a,b)),f(X,Y)).$

$X=a; Y=b$

$\text{Term} =.. [\text{likes,david,play}].$

$\text{Term} = \text{likes}(\text{david,play})$

$s([1,4,5,6]) =.. \text{List}.$

$\text{List} = [s,[1,4,5,6]].$

# Metalogische Konstrukte

- Zugriff auf PROLOG Programme
- Mengenprädikate (all solutions)
- Negation
- Cut

# Zugriff auf PROLOG Programme

clause(?Kopf,?Rumpf).

assert(+Term).

asserta(+Term).

assertz(+Term).

retract(+Term).

# Simulation globaler (binärer) Variablen

`set_flag(Flag) :- nonvar(Flag), clause(Flag, _).`

`set_flag(Flag) :- nonvar(Flag), asserta(Flag).`

`rem_flag(Flag) :- nonvar(Flag), retract(Flag).`

**Verhaltensänderung durch `assert/retract`**

# Aufruf von Goals

```
call(+Goal).
```

## Beispiel aus einem interaktiven Programm

```
ask_and_exec :-
```

```
    write('Gib mir ein Ziel '),
```

```
    read(Term),
```

```
    call(Term).
```

# Mengenprädikate (all solutions)

`findall(+Term,+Ziel,-Liste).`

## Eigenschaften

- Liste enthält alle Instanzen von Term, für die Ziel beweisbar ist.
- Für jeden erfolgreichen Beweis eine Instanz
- Liste enthält u.U. identische Instanzen
- Falls keine Lösung existiert, dann wird bei `findall/3` `Liste = []`.

# Mengenprädikate

$\text{bagof}(+ \text{Term}, + \text{Ziel}, - \text{Liste})$ .

## Eigenschaften

- Liste ist eine Sequenz von Instanzen von Term, für die Ziel beweisbar ist
- Für jeden erfolgreichen Beweis eine Instanz
- Liste enthält u.U. identische Instanzen
- Falls keine Lösung existiert, dann  $\text{failed bagof}/3$ .

# Mengenprädikate

setof(+Term,+Ziel,-Menge).

## Eigenschaften

- Menge enthält alle Instanzen von Term, für die Ziel beweisbar ist.
- Menge ist eine geordnete MENGE (keine identischen Elemente)

# Repräsentation von Wissen

- Repräsentation von “positiven” Wissen (Was gilt!)
- KEINE Möglichkeit “negatives” Wissen darzustellen
- Wie kann mit negativem Wissen umgegangen werden?

## Annahme der Weltabgeschlossenheit Closed World Assumption

- DB enthält alle Information über die Welt
- Alles, was nicht explizit notiert ist, gilt nicht.

# Negation as Failure

## Negation als Fehlschlag

Wenn auf eine variablenfreie Anfrage  $Z$  die Antwort No erfolgt, dann gilt  $\text{not } Z$

$\text{not}(+Z)$

- $\text{not } Z$  ist genau dann wahr, wenn  $Z$  endlich fehlschlägt
- $\text{not } Z$  ist genau dann falsch, wenn  $Z$  gelingt

**Cut !**

**Zweck**

Beschneiden von Suchbäumen

**Ziel**

Effizienzgewinn

## Beispiel: Cut

$\text{max}(X, Y, X) :- X \geq Y.$

$\text{max}(X, Y, Y) :- X < Y.$

$\text{max1}(X, Y, X) :- X \geq Y, !.$

$\text{max1}(X, Y, Y) :- X < Y.$

# Wirkungsweise

Klausel  $C = P : -G_1, \dots, G_k, !, G_{k+2}, \dots, G_n$

Anfrage :  $-Z$

1.  $Z$  unifiziert mit  $P$
2.  $G_1, \dots, G_k$  gelingen
3.  $!$  erfolgreich
4. Durch  $!$  wird das Programm auf die Wahl von  $C$  für das Lösen von  $Z$  festgelegt.
  - (a) Alle Alternativen zu  $C$  werden ignoriert.
  - (b)  $G_{k+2}, \dots, G_n$  werden “normal” abgearbeitet.  
Falls ein  $G_i$  ( $i > k$ ) scheitert, dann Backtracking höchstens bis zum  $!$
  - (c) Backtracking erreicht  $!$ ,  $!$  scheitert, weiter an dem Punkt, bevor  $C$  zur Lösung von  $Z$  ausgewählt wurde.

## Wirkungsweise

- Ein Cut schneidet alle Klauseln mit demselben Prädikat aus dem Suchraum, die unter der aktuellen Klausel mit dem Cut stehen.
- Ein Cut schneidet alle alternativen Lösungen der Goals aus dem Suchraum, die in der Klausel links vom Cut stehen. D.h. für Goals links vom Cut wird höchstens eine Lösung betrachtet.
- Ein Cut beeinflusst nicht die Goals zu seiner Rechten.

## Beachte

Setze einen Cut sobald Du weißt, daß dies die richtige Klausel ist – nicht später, aber auch nicht früher.

Verzögere Ausgabe-Unifikationen bis nach dem Cut.

# Beispiel

$\text{max1}(X, Y, X) :- X \geq Y, !.$

$\text{max1}(X, Y, Y) :- X < Y.$

$\text{max1}(X, Y, X) :- X \geq Y, !.$

$\text{max1}(X, Y, Y).$

$\text{max1}(X, Y, Z) :- X \geq Y, !, Z = X.$

$\text{max1}(X, Y, Y).$

# Red Cuts

$\text{member1}( X, [X|Y] ).$

$\text{member1}( X, [_|Y] ) :- \text{member1}(X,Y).$

$\text{member2}( X, [X|Y] ) :- !.$

$\text{member2}( X, [_|Y] ) :- \text{member2}(X,Y).$

# Cut und Fail

$\text{not } G :- G, !, \text{fail.}$

$\text{not } G.$

$\text{not\_unify}(X,Y) :- \text{unify}(X,Y), !, \text{fail.}$

$\text{not\_unify}(-,-).$