Prolog-Arithmetik

Verfügbare arithmetische Operationen („reelle“ Zahlen):
+, - , *, /, //, mod und ggf. weitere

Verfügbare arithmetische Vergleichsoperationen:
> , < , >=, =<, =:= , =/= (und ggf. weitere)

Vergleiche für numerisch auswertbare Ausdrücke auf beiden Seiten

Unterschied:

`is` erlaubt Ausdruck nur rechts,
ggf. Unifizierung mit Variabler auf linker Seite

`=:=` überprüft Identität zweier Ausdrücke

Prolog-Arithmetik

Ursprüngliche Definition

```
factorial(o,s(o)).
factorial(s(N),F) :- factorial(N,F1), mult(s(N),F1,F).
```

erlaubt Anfrage

```
?- factorial(X,Y).
```

Bei Prolog-Arithmetik dagegen nicht möglich

```
factorial(0,1).
factorial(N,F) :- N> 0, N1 is N-1, factorial(N1,F1), F is N * F1.
```

- Laufzeitfehler, wenn Ausdrücke nicht auswertbar sind

Speziell bei ungebundenen Variablen:
Zwang zu „prozeduraler“ Reihenfolge der subgoals

Prolog-Arithmetik

Ursprüngliche Definition

```
add(Summand1,Summand2,Summe)
```

erlaubt Definition

```
minus(Minuend,Subtrahend,Differenz) :- add(Subtrahend,Differenz,Minuend)
```

Bei Verwendung der Prolog-Arithmetik ist das nicht möglich.

```
addiere(X,Y,S) :- S is X + Y
```

entspricht nicht früherem `add(X,Y,S)`

Operatoren

Standard-Schreibweise in Prolog:
Struktur/Term: funktor(Argumente)
Alternativ: Operator-Schreibweise für bessere Lesbarkeit

		Funktor
Infix-Operator	7 + 9 für Struktur + (7,9)	+/2
Prefix-Operator	- 9 für Struktur - (9)	-/1
Postfix-Operator	9! (Fakultät)	!/1

Zu klären:
- Priorität: $7 + 9 * 2 = 7 + (9 * 2)$
- Assoziativität: $7 - 9 - 2 = (7 - 9) - 2$

Operatoren

Operatoren deklarieren mittels

`op(Priorität, Typ, Name)`

```
op( 500, yfx, '-' ).
op( 500, yfx, '+' ).
op( 400, yfx, '*' ).
```

- Vergabe von Prioritäten: 0,...,1200 (Maximum)

Priorität eines Terms: Priorität des Hauptfunktors
Priorität 0 haben:

Atome (außer Operatoren), Zahlen, Variable, Zeichenketten,
in Klammern eingeschlossene Terme

Operatoren

- Festlegung der Assoziativität durch Typen:
  - Infixoperatoren `xfx, xfy, yfx`,
  - Präfixoperatoren `fx, fy`,
  - Postfixoperatoren `xf, yf`.

```
op( 500, yfx, '-' ).
op( 500, yfx, '+' ).
op( 400, yfx, '*' ).
```

`f`: Funktor
`x`: Term mit geringerer Priorität als `op`
`y`: Term mit maximal gleicher Priorität wie `op`
(andernfalls Klammern notwendig)

Beispiele für Operatoren

Standardmäßig im Prolog-Interpreter (built-in-Operatoren)

1200 xfx	:-	
1200 fx	?-	
1100 xfy	;	
1000 xfy	,	
900 fy	not	
700 xfx	=, \=	(Unifikation)
	==, \==	(Identität für Terme)
	<, =:=, >, =<, >=, =\=, is	(Arithmetik)
500 yfx	+, -	
500 fx	+, -,	
400 yfx	*, /, //,mod	

Rekursive Definitionen

```
zahl(0).
zahl(s(X)):-zahl(X).
```

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

```
erreichbar(X,X).
erreichbar(X,Y)
        :- benachbart(X,Z),erreichbar(Z,Y).
```

Transitiver Abschluss von Relationen

Logisch äquivalente rekursive Definitionen:

```
erreichbar(X,Y):- benachbart(X,Z), erreichbar(Z,Y).
erreichbar(X,X).
```

```
erreichbar(X,X).
erreichbar(X,Y):- benachbart(X,Z), erreichbar(Z,Y).
```

```
erreichbar(X,X).
erreichbar(X,Y):- erreichbar(Z,Y), benachbart(X,Z).
```

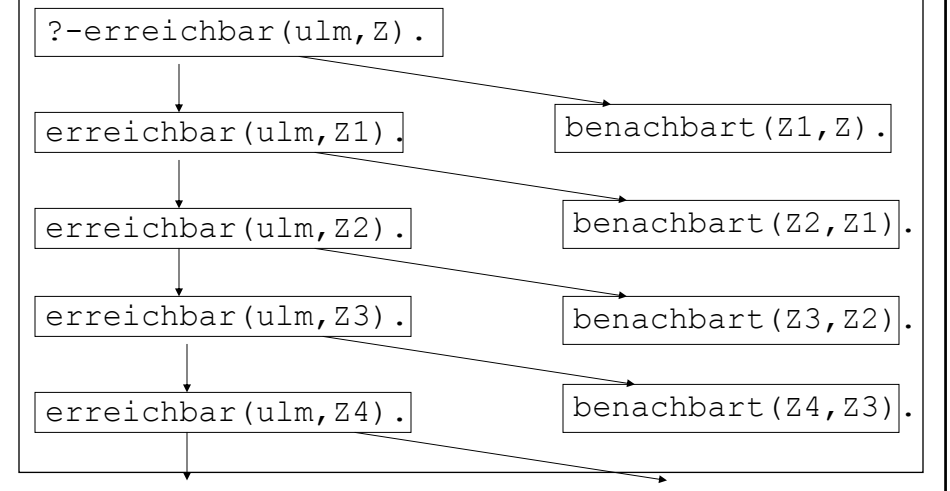
```
erreichbar(X,Y):- erreichbar(Z,Y), benachbart(X,Z).
erreichbar(X,X).
```

Inhaltlich äquivalent z.B. auch:

```
erreichbar(X,Y) :- erreichbar(X,Z), benachbart(Z,Y).
erreichbar(X,X).
```

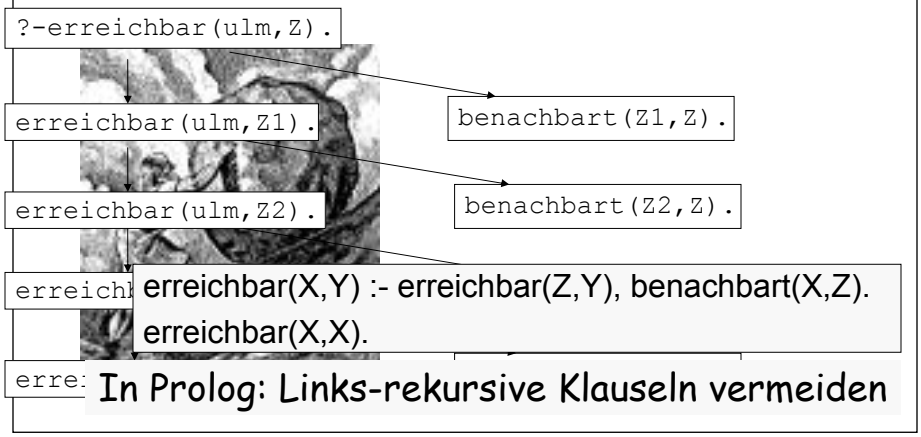

Unendliche Beweisversuche

```
erreichbar(X,Y):-erreichbar(X,Z),benachbart(Z,Y).
```



Deklarative vs.prozedurale Semantik

Unterschiedliche Resultate bei
deklarativer und prozeduraler Semantik



In Prolog: Links-rekursive Klauseln vermeiden

Deklarative vs.prozedurale Semantik

Januskopf von Prolog:
Unterschiedliche Resultate
bei deklarativer und prozeduraler Semantik

- Reihenfolge der Beweisversuche
  (Auswirkungen z.B. auf rekursive Klauseln, Negation)
- (zusätzliche) Arithmetik
- Abhängigkeit von „Seiteneffekten“,
  z.B. Eingabe-/Ausgabe-Abhängigkeit
- Eingriffe in Beweisversuche (cut)
- Meta-logische Prädikate
- Programm-Modifikation

Eingriff in die Abarbeitung: Cut

Verändern der Suchstrategie: Prädikat `!/0` (*Cut*)

`!/0` gelingt stets und löscht Choice-Points für
- aktuelle Klausel
- subgoals im Klauselkörper, die vor dem Cut stehen
- subgoals dieser subgoals usw.

Folge:
Gefundene Lösung wird ,,eingefroren“
Alternativen für Backtracking entfallen

Eingriff in die Abarbeitung: Cut

```
prinz(X):-grandchild(X,cronus),male(X).
```

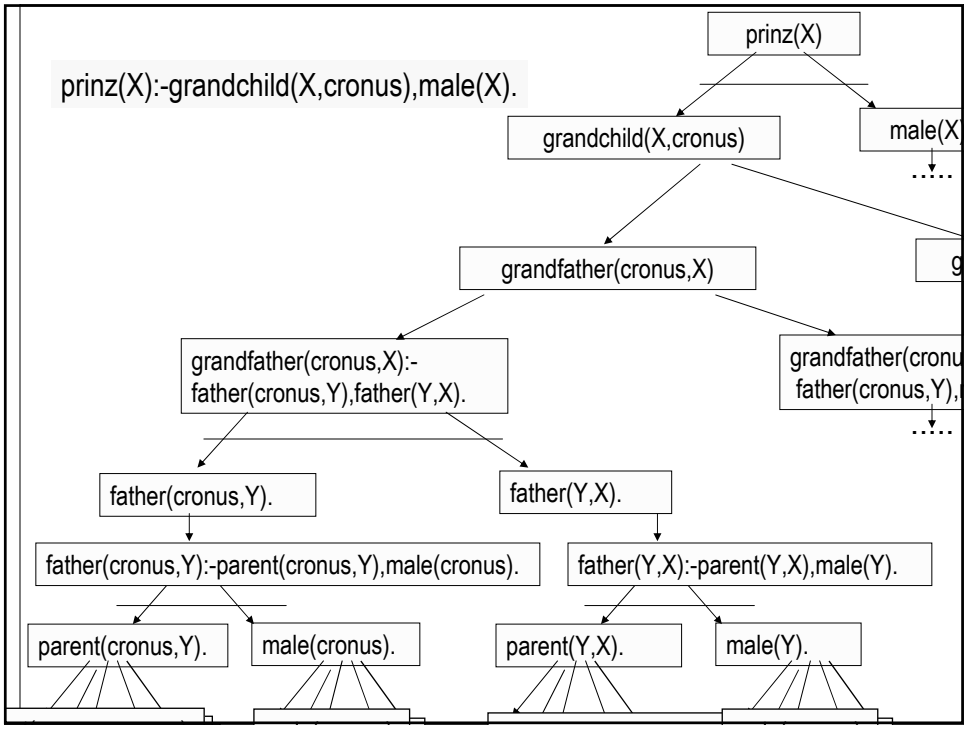
```
?- prinz(X).
X = hermes ;
X = hephaestus ;
X = apollo ;
X = hephaestus ;
no
```

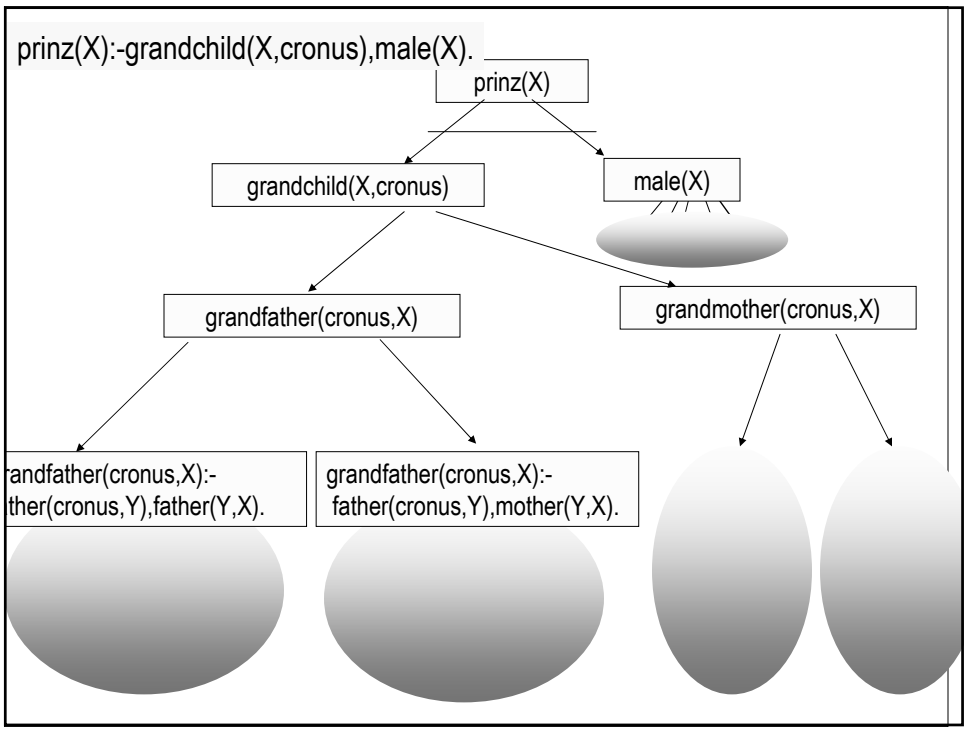
```
kronprinz(X):-grandchild(X,cronus),!,male(X).
```

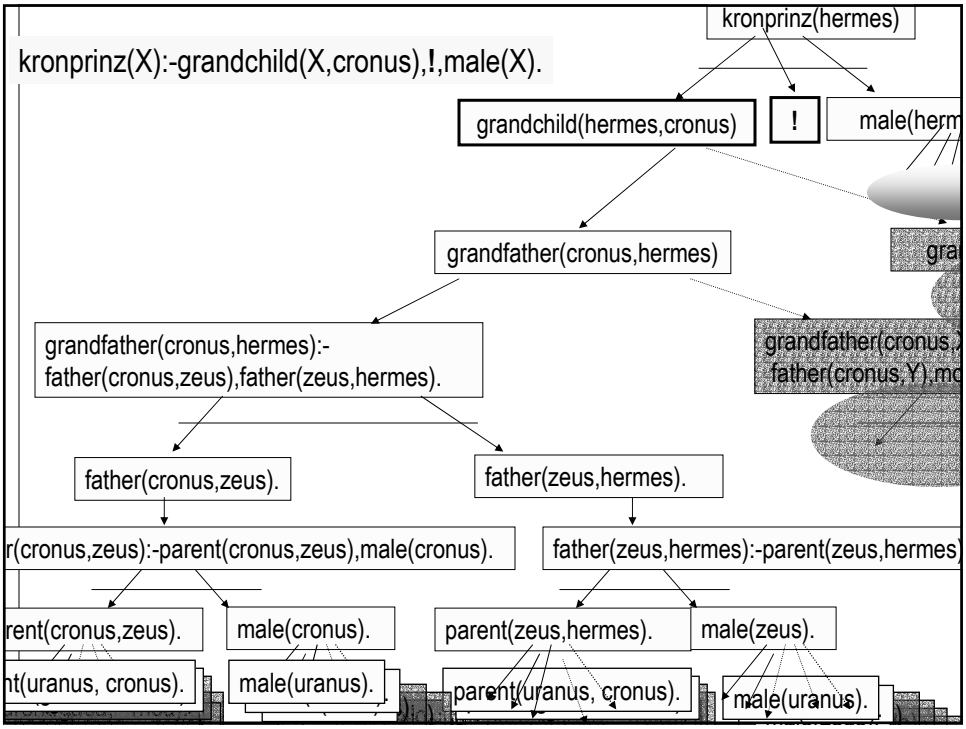
```
?- kronprinz(X).
X = hermes ;
no
```

Worin besteht der inhaltliche Unterschied:

```
kronprinz(X):-grandchild(X,cronus),male(X), ! .
```


prinz(X):-grandchild(X,cronus),male(X).
prinz(X)
grandchild(X,cronus)
male(X)
grandfather(cronus,X)
grandfather(cronus,X):-
father(cronus,Y),father(Y,X).
father(cronus,Y).
father(Y,X).
father(cronus,Y):-parent(cronus,Y),male(cronus).
father(Y,X):-parent(Y,X),male(Y).
parent(cronus,Y).
male(cronus).
parent(Y,X).
male(Y).


prinz(X):-grandchild(X,cronus),male(X).
prinz(X)
grandchild(X,cronus)
male(X)
grandfather(cronus,X)
grandmother(cronus,X)
grandfather(cronus,X):-
father(cronus,Y),mother(Y,X).



Cut zur Beschleunigung

(unnötige) Alternativen vermeiden

```
maximum1(X,Y,X) :-  X >= Y, !.
maximum1(X,Y,Y) :-  X <  Y .
```

Grüner cut: gleiches Resultat ohne Cut

```
maximum2(X,Y,X) :- X >= Y, !.
maximum2(X,Y,Y).
```

Roter cut: anderes Resultat ohne Cut

Fallunterscheidung

```
case(...)  :-  condition-1(...),declaration-1(...).
case(...)  :-  condition-2(...),declaration-2(...).
...
case(...)  :-  condition-n(...),declaration-n(...).
```

- Mit Backtracking, ggf. weitere Regel bearbeiten

```
case(Geld,Essen) :- Geld>500,adlon(Essen).
case(Geld,Essen) :- Geld>50,steakhouse(Essen).
case(Geld,Essen) :- Geld>5,doener(Essen).
case(Geld,Essen) :- selberkochen(Essen)
```


Fallunterscheidung mit cut

```
case(...) :- condition-1(...), !, declaration-1(...).
case(...) :- condition-2(...), !, declaration-2(...).
...
case(...) :- condition-n(...), !, declaration-n(...).
```

- Ohne Backtracking, höchstens eine Regel bearbeiten

```
case(Geld,Einladung) :- Geld>500, !, adlon(Einladung).
case(Geld,Einladung) :- Geld>50,   !, steakhouse(Einladung).
case(Geld,Einladung) :- Geld>5,    !, doener(Einladung).
case(Geld,Einladung) :-            !, selberkochen(Einladung)
```

„If then else"

CWA (2)

Wann soll Interpreter Antwort „no“ auf Anfrage Q liefern?

Logische Varianten:

1. Wenn $\neg$Q bewiesen wurde.
2. Wenn Q nachweisbar nicht bewiesen werden kann.
3. Wenn alle Beweisversuche für Q fehlgeschlagen sind.

Dabei jeweils implizite Annahmen bzgl. Negation.

Nicht-logische Varianten:

I) Wenn die verfügbaren Argumente für „nicht Q“ sprechen.
II) Wenn die verfügbaren Argumente gegen Q sprechen.
III) Im Zweifelsfall für den Angeklagten ...

CWA (2)

In Prolog:

Variante 3 **„Negation by (finite) failure“**

Wenn alle Beweisversuche für Q fehlgeschlagen sind.

Bedeutung der Antwort „no“:
Alle Beweisversuche sind fehlgeschlagen.

Probleme:

- In PK1: Kein allgemeines Verfahren
  (Nicht-Allgemeingültigkeit ist nicht aufzählbar)
- Unterschiede zur logischen Negation