

Deskriptive Programmierung

SS 2015

Jun.-Prof. Dr. Janis Voigtländer
Institut für Informatik III
Universität Bonn

Deskriptive Programmierung

Motivation/Einführung Logisches Programmieren

„An algorithm can be regarded as consisting of a logic component, which specifies the knowledge to be used in solving problems, and a control component, which determines the problem-solving strategies by means of which that knowledge is used. The logic component determines the meaning of the algorithm whereas the control component only affects its efficiency.

The efficiency of an algorithm can often be improved by improving the control component without changing the logic of the algorithm. We argue that computer programs would be more often correct and more easily improved and modified if their logic and control aspects were identified and separated in the program text. “

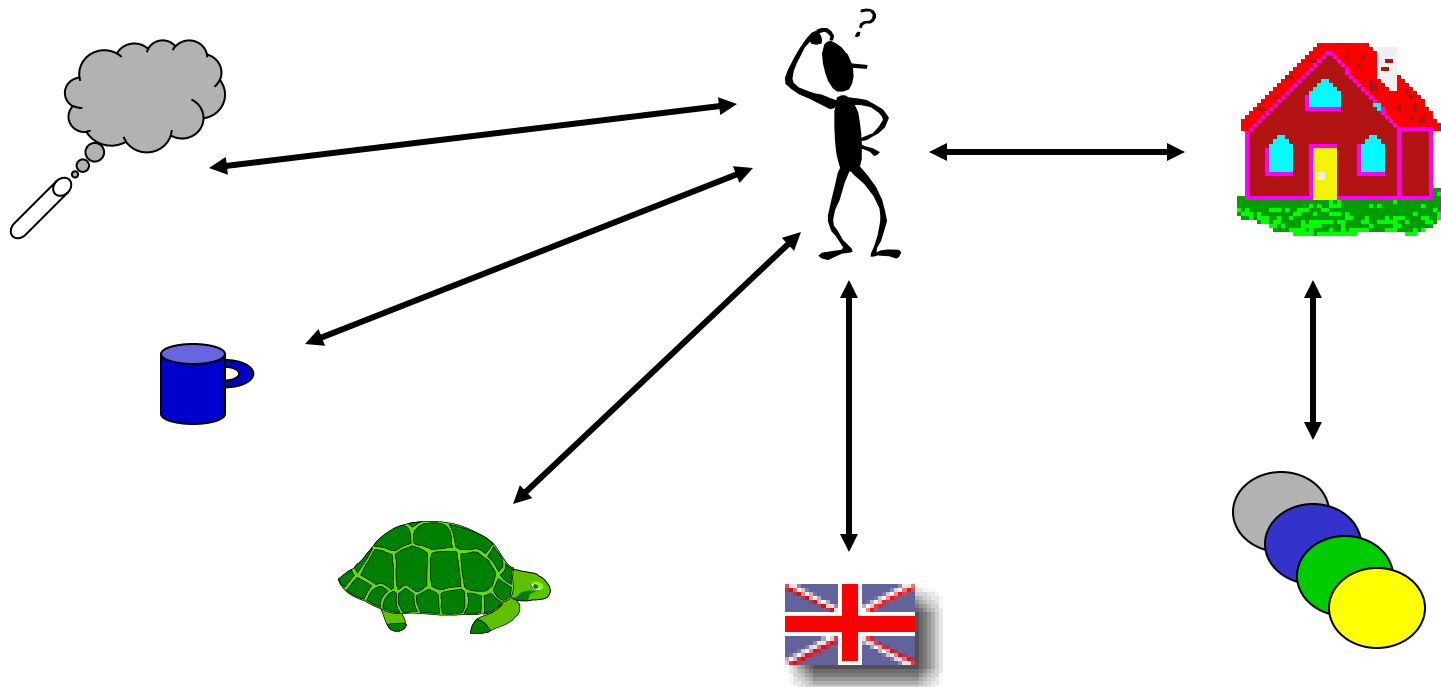
Robert Kowalski, 1979

- **Prolog** steht für “Programming with logic”.
- Es ist die am **weitesten verbreitete** logische Programmiersprache.
- Ein wenig Geschichte zu Prolog:
 - 1965: John Alan Robinson legt theoretische Grundlagen für Theorembeweiser mit dem Resolutionskalkül.
 - 1972: Alain Colmerauer (Marseilles) und seine Gruppe entwickeln Prolog.
 - Mitte 70er: David D.H. Warren baut den ersten lauffähigen Compiler, wonach sich der spätere DEC-10 Standard richtet (Edinburgh-Standard).
 - 1981–92: 5th Generation Computer Project in Japan (machte Prolog bekannt)

Ein berühmtes logisches Puzzle als deskriptiv spezifiziertes Problem

„There are five **houses**, each of a different **color** and inhabited by a man of a different **nationality** with a different **pet**, **drink** and brand of **smokes** ...“

(das „Zebra Puzzle“ oder „Einstein's Riddle“, s. http://en.wikipedia.org/wiki/Zebra_Puzzle)



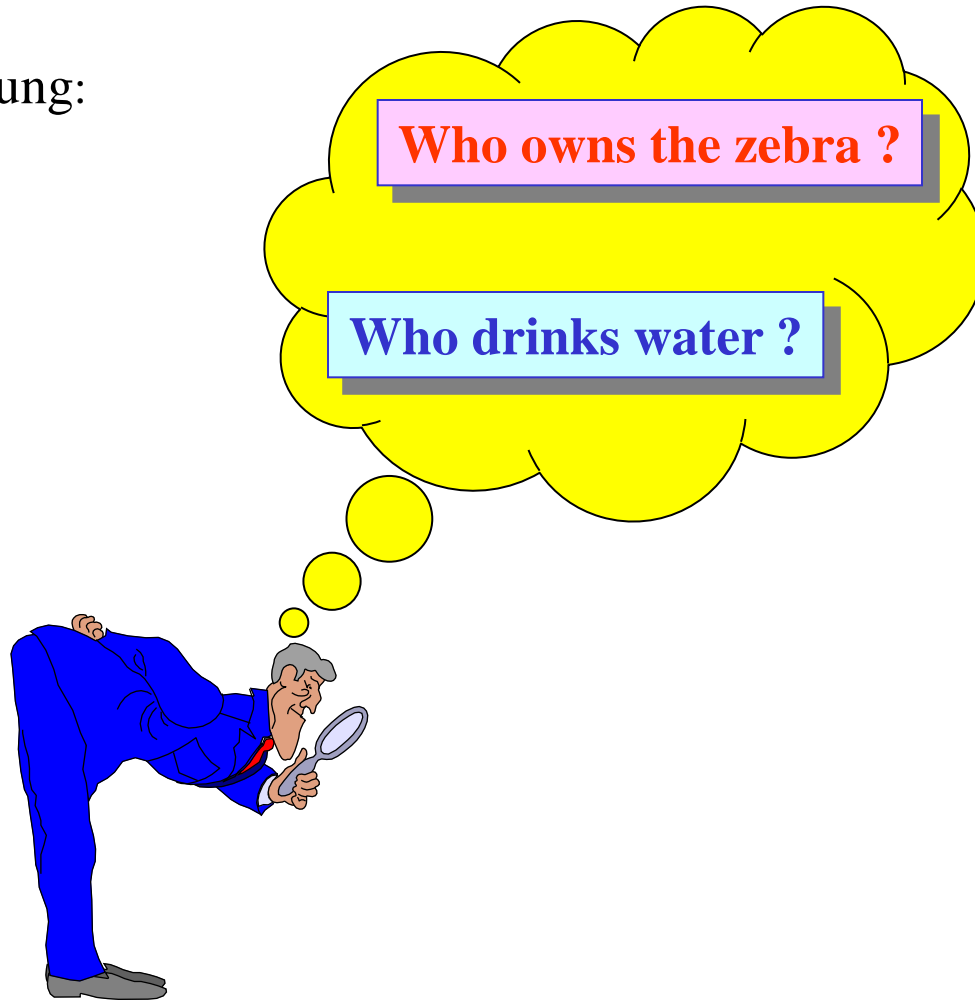
Puzzle (1)

Insgesamt gelten **14 Regeln** (engl. „clues“), die die „Welt“ des Puzzles definieren:

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.
3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.
5. The green house is immediately to the right of the ivory house.
6. The Winston smoker owns snails.
7. Kools are smoked in the yellow house.
8. Milk is drunk in the middle house.
9. The Norwegian lives in the leftmost house.
10. The man who smokes Chesterfield lives in the house next to the man with the fox.
11. Kools are smoked in the house next to the house where the horse is kept.
12. The Lucky Strike smoker drinks orange juice.
13. The Japanese smokes Parliaments.
14. The Norwegian lives next to the blue house.

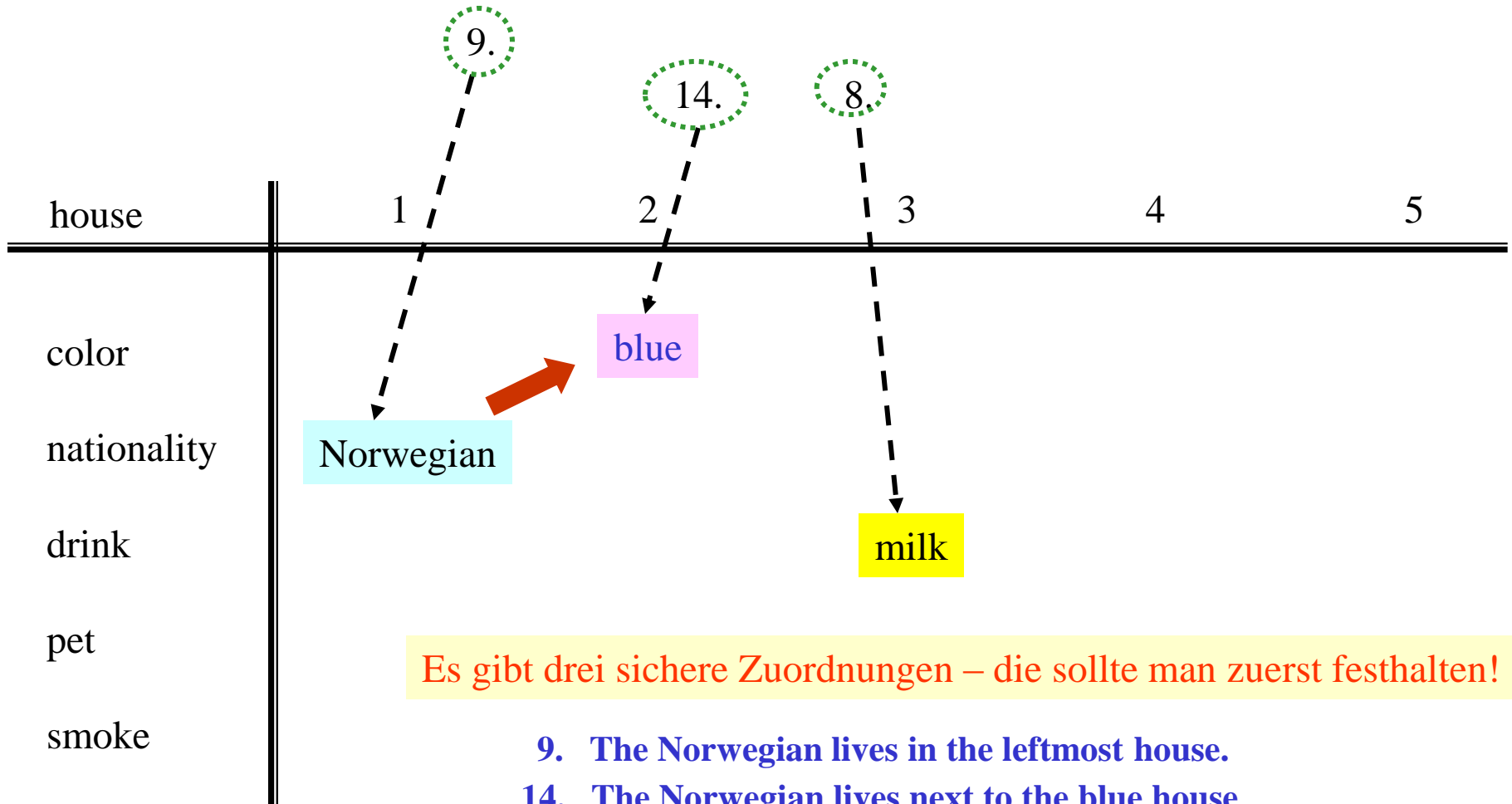
Puzzle (2)

Problemstellung:



Puzzle (3)

Systematische Konstruktion der Lösung (durch den Menschen):



Es gibt drei sichere Zuordnungen – die sollte man zuerst festhalten!

- 9. The Norwegian lives in the leftmost house.
- 14. The Norwegian lives next to the blue house.
- 8. Milk is drunk in the middle house.

Puzzle (4)

Bedingung 5:

5. The green house is immediately to the right of the ivory house.

... lässt nur zwei Möglichkeiten offen:

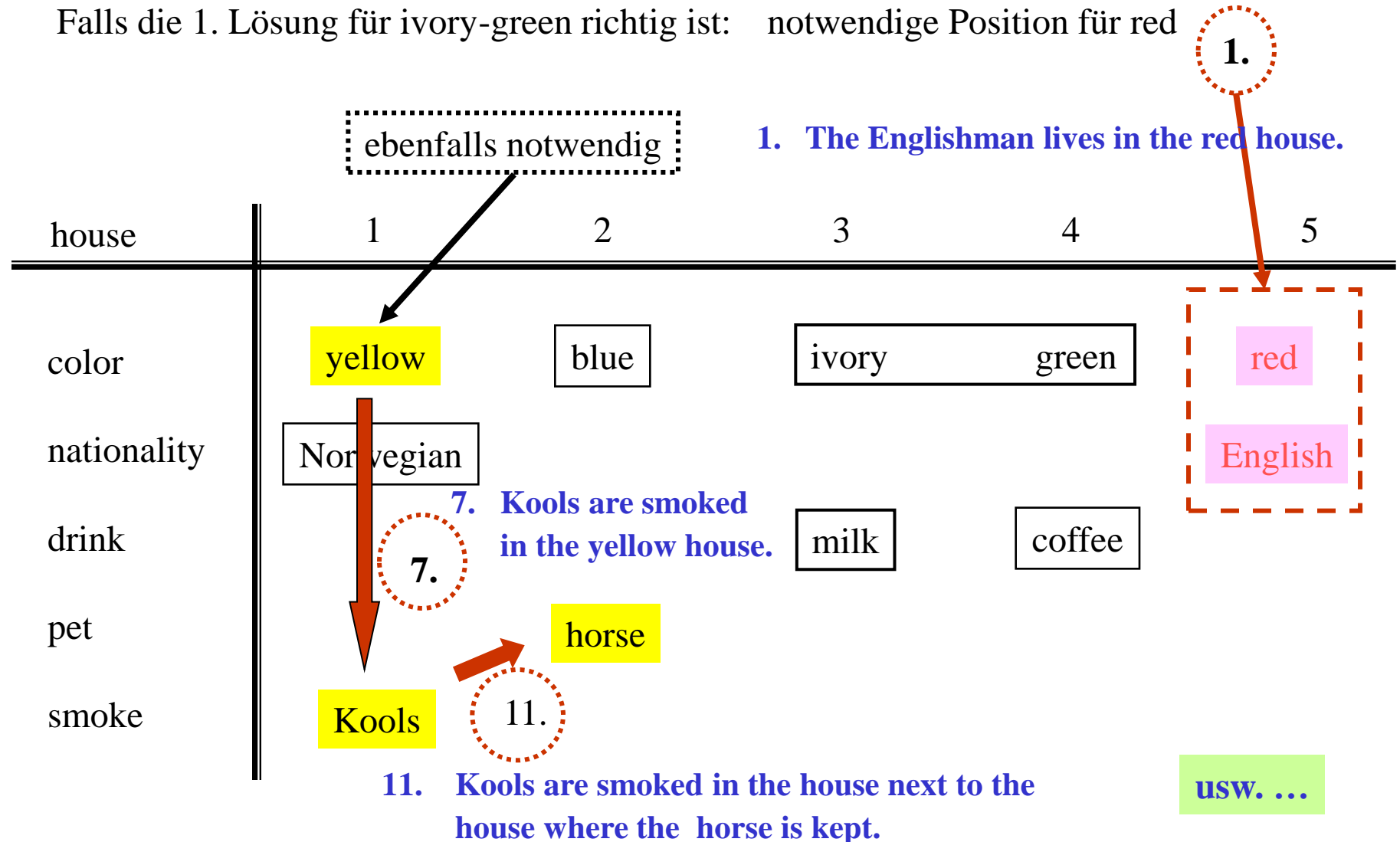
house	1	2	3	4	5
color		blue	ivory	green	green
nationality	Norwegian				
drink			milk	coffee	coffee
pet					
smoke					

Eine direkte Konsequenz daraus wäre:

3. Coffee is drunk in the green house.

Puzzle (5)

Falls die 1. Lösung für ivory-green richtig ist: notwendige Position für red



Puzzle (6)

Eindeutige Lösung des Rätsels (zu finden durch diverse Backtrackingschritte):

house	1	2	3	4	5
color	yellow	blue	red	ivory	green
nationality	Norwegian	Ukrainian	English	Spanish	Japanese
drink	water	tea	milk	juice	coffee
pet	fox	horse	snails	dog	zebra
smoke	Kools	Chesterfield	Winston	Lucky Strike	Parliaments

Puzzle: eine mögliche Spezifikation in Prolog

```
right_of(R, L, [ L | [ R | _ ] ).
```

```
right_of(R, L, [ _ | Rest ] ) :- right_of(R, L, Rest).
```

```
next_to(X, Y, List) :- right_of(X, Y, List).
```

```
next_to(X, Y, List) :- right_of(Y, X, List).
```

```
zebra(Zebra_Owner) :-
```

- 8. ^ 9.** Houses = [[_, norwegian, _, _, _], _, [_, _, milk, _, _], _, _],
- 1.** member([red, englishman, _, _, _], Houses),
- 2.** member([_, spaniard, _, dog, _], Houses),
- 3.** member([green, _, coffee, _, _], Houses),
- 4.** member([_, ukrainian, tea, _, _], Houses),
- 5.** right_of([green, _, _, _], [ivory, _, _, _], Houses),
- 6.** member([_, _, _, snails, winston], Houses),
- 7.** member([yellow, _, _, _, kools], Houses),
- 10.** next_to([_, _, _, chesterfield], [_, _, _, fox, _], Houses),
- 11.** next_to([_, _, _, kools], [_, _, _, horse, _], Houses),
- 12.** member([_, _, juice, _, lucky], Houses),
- 13.** member([_, japanese, _, _, parliaments], Houses),
- 14.** next_to([_, norwegian, _, _, _], [blue, _, _, _], Houses),
- ?** member([_, Zebra_Owner, _, zebra, _], Houses),
- ?** member([_, _, water, _, _], Houses).

Deskriptive Programmierung

Prolog-Grundlagen/Syntax

Prolog im einfachsten Fall: Fakten und Anfragen

- Eine Art Datenbank mit einer Reihe von Fakten:

```
woman(mia) .  
woman(jody) .  
woman(yolanda) .  
playsAirGuitar(jody) .
```

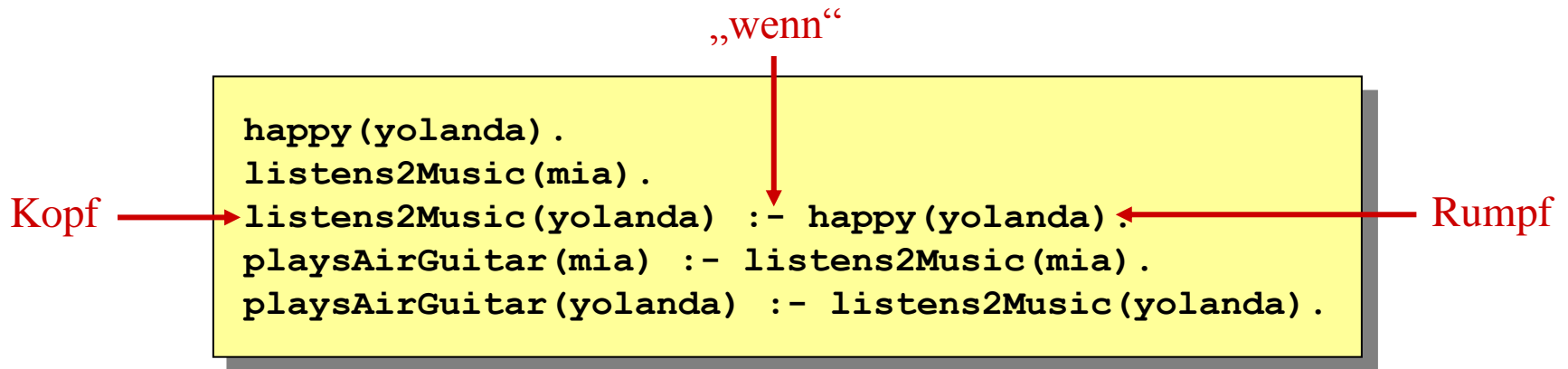
- Anfragen:

```
?- woman(mia) .  
true.  
  
?- playsAirGuitar(jody) .  
true.  
  
?- playsAirGuitar(mia) .  
false.  
  
?- playsAirGuitar(vincent) .  
false.  
  
?- playsPiano(jody) .  
false.
```

← Punkt wichtig!

← oder Fehlermeldung

Fakten + einfache Implikationen



- Anfragen:

```
?- playsAirGuitar(mia).  
true.  
  
?- playsAirGuitar(yolanda).  
true.
```

wegen:

```
happy(yolanda)  
⇒ listens2Music(yolanda)  
⇒ playsAirGuitar(yolanda)
```

```
happy(vincent) .  
listens2Music(butch) .  
playsAirGuitar(vincent) :- listens2Music(vincent) ,  
                             happy(vincent) .  
playsAirGuitar(butch) :- happy(butch) .  
playsAirGuitar(butch) :- listens2Music(butch) .
```

„und“

Alternativen →

- Anfragen:

```
?- playsAirGuitar(vincent) .  
false.  
  
?- playsAirGuitar(butch) .  
true.
```

- alternative Schreibweise:

```
...  
playsAirGuitar(butch) :- happy(butch) ;  
                        listens2Music(butch) .
```

„oder“

```
woman(mia) .  
woman(jody) .  
woman(yolanda) .  
  
loves(vincent,mia) .  
loves(marsellus,mia) .  
loves(mia,vincent) .  
loves(vincent,vincent) .
```

mehrstelliges Prädikat

- Anfragen:

```
?- woman(X) .  
X = mia ;  
X = jody ;  
X = yolanda.  
  
?- loves(vincent,X) .  
X = mia ;  
X = vincent.  
  
?- loves(vincent,X) , woman(X) .  
X = mia ;  
false.
```

vom Nutzer einzugeben

Variablen in Regeln (nicht nur in Anfragen)

```
loves(vincent,mia).  
loves(marsellus,mia).  
loves(mia,vincent).  
  
jealous(X,Y) :- loves(X,Z), loves(Y,Z).
```

- Anfragen:

```
?- jealous(marsellus,X).  
X = vincent ;  
X = marsellus ;  
false.  
  
?- jealous(X,_).  
X = vincent ;  
X = vincent ;  
X = marsellus ;  
X = marsellus ;  
X = mia.
```

→ anonyme Variable

Variablen in Regeln (nicht nur in Anfragen)

```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

jealous(X,Y) :- loves(X,Z), loves(Y,Z), X \= Y.
```

- Anfragen:

```
?- jealous(marsellus,X).
X = vincent ;
false.

?- jealous(X,_).
X = vincent ;
X = marsellus ;
false.

?- jealous(X,Y).
X = vincent,
Y = marsellus ;
X = marsellus,
Y = vincent ;
false.
```

wichtig dass am Ende

Einige Beobachtungen zu Variablen

```
loves (vincent, mia) .  
loves (marsellus, mia) .  
loves (mia, vincent) .  
  
jealous (X, Y) :- loves (X, Z) , loves (Y, Z) , X \= Y.
```

- Variablen in Regeln und Anfragen sind unabhängig voneinander.

```
?- jealous (marsellus, X) .  
X = vincent ;  
false.
```

- Innerhalb einer Regel oder Anfrage stehen gleiche Variablen für gleiche Objekte.
- Aber verschiedene Variablen stehen nicht notwendigerweise für verschiedene Objekte.
- **Es sind auch mehrfache Vorkommen der gleichen Variable im Kopf möglich!**
- **In Regeln können im Rumpf Variablen auftauchen, die nicht im Kopf vorkommen!**


```
loves (vincent, mia) .  
loves (marsellus, mia) .  
loves (mia, vincent) .  
  
jealous (X, Y) :- loves (X, Z) , loves (Y, Z) , X \= Y.
```

- Was ist die „logische“ Interpretation von **Z** oben? (bzw. der gesamten Regel?)
- Denkbar, für beliebige (aber feste) **X** , **Y**:
wenn für jede Wahl von **Z** gilt: **loves (X, Z)** , und **loves (Y, Z)** , und **X \= Y** ,
dann gilt auch: **jealous (X, Y)**
- Oder, für beliebige (aber feste) **X** , **Y**:
für jede Wahl von **Z** gilt: wenn **loves (X, Z)** , und **loves (Y, Z)** , und **X \= Y** ,
dann gilt auch: **jealous (X, Y)**

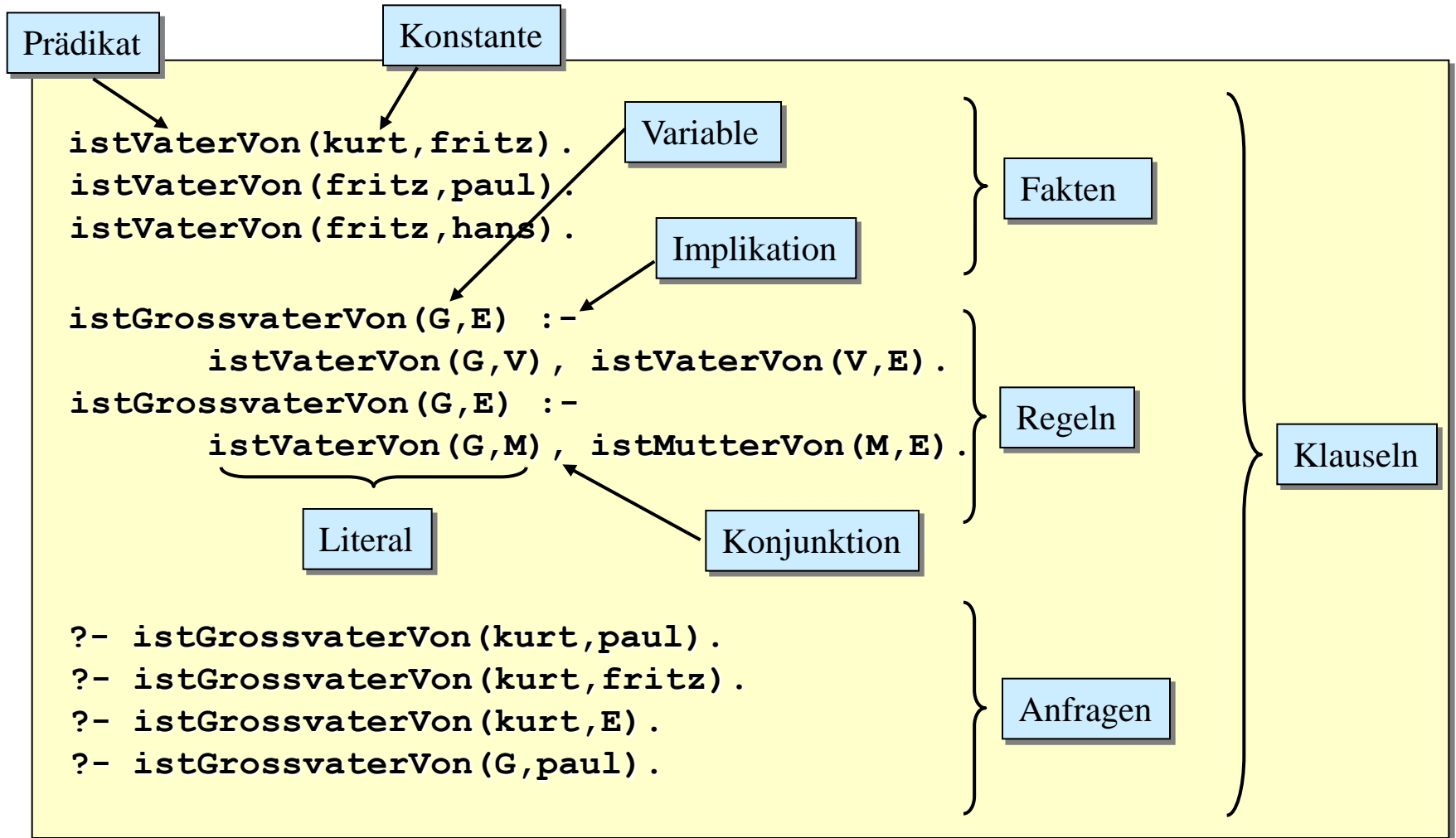
???

```
loves (vincent, mia) .  
loves (marsellus, mia) .  
loves (mia, vincent) .  
  
jealous (X, Y) :- loves (X, Z) , loves (Y, Z) , X \= Y.
```

- Was ist die „logische“ Interpretation von **Z** oben? (bzw. der gesamten Regel?)
- Denkbar, für beliebige (aber feste) **X** , **Y**:
wenn für jede Wahl von **Z** gilt: **loves (X, Z)** , und **loves (Y, Z)** , und **X \= Y** ,
dann gilt auch: **jealous (X, Y)**
- Oder, für beliebige (aber feste) **X** , **Y**:
für jede Wahl von **Z** gilt: wenn **loves (X, Z)** , und **loves (Y, Z)** , und **X \= Y** ,
dann gilt auch: **jealous (X, Y)**

```
loves (vincent, mia) .  
loves (marsellus, mia) .  
loves (mia, vincent) .  
  
jealous (X, Y) :- loves (X, Z) , loves (Y, Z) , X \= Y.
```

- Was ist die „logische“ Interpretation von **Z** oben? (bzw. der gesamten Regel?)
- Oder, für beliebige (aber feste) **X** , **Y**:
für jede Wahl von **Z** gilt: wenn **loves (X, Z)** , und **loves (Y, Z)** , und **X \= Y** ,
dann gilt auch: **jealous (X, Y)**
- Logisch äquivalent, für beliebige (aber feste) **X** , **Y**:
wenn für irgendeine Wahl von **Z** gilt: **loves (X, Z)** , und **loves (Y, Z)** , und **X \= Y** ,
dann gilt auch: **jealous (X, Y)**



- Zum Aufbau von Klauseln verwendet Prolog verschiedene Objekte.
- Diese unterteilen sich in:
 - **Konstanten** (Zahlen, Zeichenfolgen, ...)
 - **Variablen** (X,Y, InLand, ...)
 - **Operatortermine** (... 1 + 3 * 4 ...)
 - **Strukturen** (datum(27,11,2007), person(fritz, mueller), ...
zusammengesetzt, rekursiv, „unendlich“, ...)
- **Achtung**: Prolog hat kein Typsystem

Konstanten in Prolog

- **Zahlen**

```
-17 -2.67e+021 0 1 99.9 512
```

- **Atome**, d.h. Zeichenfolgen, die einer der folgenden drei Regeln genügen:

1. Die Zeichenfolge beginnt mit einem Kleinbuchstaben, gefolgt von beliebig vielen Klein- und Großbuchstaben, Ziffern und Unterstrichen '_'.
Beispiel: `fritz`, `new_york`
2. Die Zeichenfolge beginnt und endet mit einem Apostroph ('). Dazwischen können beliebige Zeichen stehen. Soll ein Apostroph selbst in der Zeichenkette vorkommen, muss er doppelt angegeben werden.
Beispiel: `'I don''t know!'`
3. Die Zeichenfolge besteht nur aus Sonderzeichen.
Beispiel: `new-york_xyz`, `123`

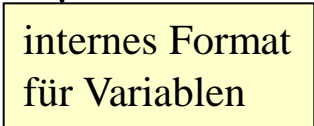
richtig:	<code>fritz</code>	<code>new_york</code>	<code>:-</code>	<code>--></code>	<code>'I don''t know!'</code>
falsch:	<code>Fritz</code>	<code>new-york</code>	<code>_xyz</code>	<code>123</code>	

Variablen in Prolog

- **Variablen:**

- Name beginnt mit einem Großbuchstaben oder einem Unterstrich '_'.

- Beispiele: `Land Jahr M V _45 _G107 _europa`



internes Format
für Variablen

- Anonyme Variablen (Darstellung mit lediglich '_'):

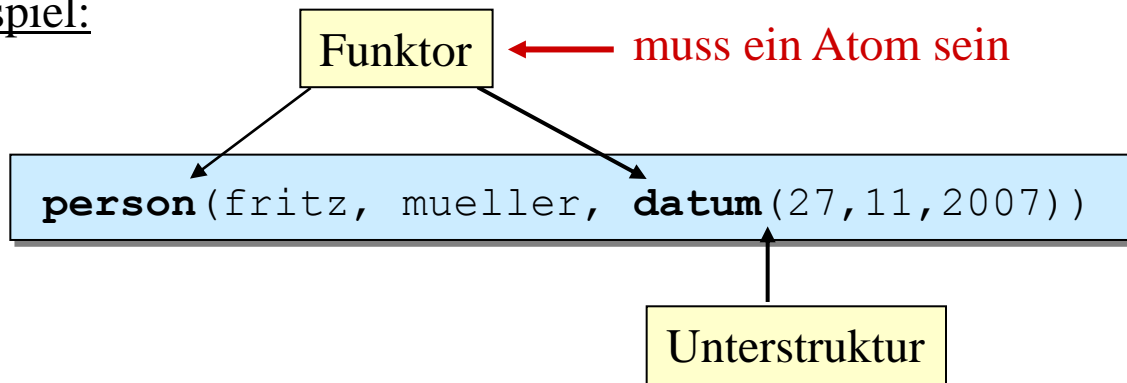
- wenn das Objekt nicht interessiert:

```
?- istVaterVon(_,fritz).
```

Strukturen in Prolog

- **Strukturen** repräsentieren Objekte, die aus mehreren anderen Objekten zusammengesetzt sind.

- Beispiel:



Funktoren: `person/3`, `datum/3`

- Dadurch Modellierung von algebraischen Datentypen – aber keine Typisierung. Somit wäre `person(1, 2, 'a')` auch eine legale Struktur.
- Beliebige **Schachtelungstiefe** ist erlaubt – im Prinzip unendlich.

Vordefinierte Syntax für spezielle Strukturen:

- Es gibt einen vordefinierten „Listentyp“ als rekursive Datenstruktur:

```
[1,2,a] [1|[2,a]] [1,2|[a]] [1,2|. (a, [])] .(1, .(2, .(a, [])))
```

- Zeichenketten werden als Listen von ASCII-Codes dargestellt:

```
"Prolog" = [80, 114, 111, 108, 111, 103]  
          = .(80, .(114, .(111, .(108, .(111, .(103, [ ])))))
```

Operatoren:

- Operatoren sind Funktoren in Operatorschreibweise.
- Beispiel: arithmetische Ausdrücke
 - Mathematische Funktionen sind als Operatoren definiert.

• `1 + 3 * 4` ist als folgende Struktur zu sehen: `+ (1, * (3, 4))`

Einfaches Beispiel für Arbeit mit Datenstrukturen

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .
```

```
?- add(s(0), s(0), s(s(0))) .  
true.  
  
?- add(s(0), s(0), N) .  
N = s(s(0)) ;  
false.
```

- Zur Erinnerung, in Haskell:

```
data Nat = Zero | Succ Nat  
  
add :: Nat → Nat → Nat  
add Zero     m = m  
add (Succ n) m = Succ (add n m)
```

Systematischer Zusammenhang/Herleitung?

- Wesentlicher Unterschied Haskell/Prolog:

Funktionen

vs.

Prädikate/Relationen

$f\ x\ y = z$

„entspricht“

$p(x, y, z)$

- Zunächst etwas naiver Versuch, diesen Zusammenhang auszunutzen:

add Zero m = m

↓
add(Zero,m,m)

↓
add(0, x, x) .

add (Succ n) m = Succ (add n m)

↓
add(Succ n,m,Succ (add n m))

↓
???

Systematischer Zusammenhang/Herleitung?

- Wesentlicher Unterschied Haskell/Prolog:

Funktionen

vs.

Prädikate/Relationen

$f\ x\ y = z$

„entspricht“

$p(x, y, z)$.

- Systematische Vermeidung verschachtelter Aufrufe:

`add (Succ n) m = Succ (add n m)`



`add (Succ n) m = Succ m' where m' = add n m`



`add(Succ n,m,Succ m')` wenn `add(n,m,m')`



`add(s(X), Y, s(Z)) :- add(X, Y, Z).`

Zur Flexibilität von Prolog-Prädikaten

```
add(0,X,X) .  
add(s(X),Y,s(Z)) :- add(X,Y,Z) .
```

```
?- add(N,M,s(s(0))) .  
N = 0,  
M = s(s(0)) ;  
N = s(0),  
M = s(0) ;  
N = s(s(0)),  
M = 0 ;  
false.  
  
?- add(N,s(0),s(s(0))) .  
N = s(0) ;  
false.  
  
?- add(N,M,O) .
```

???

Zur Flexibilität von Prolog-Prädikaten

```
add(0,X,X) .  
add(s(X),Y,s(Z)) :- add(X,Y,Z) .  
  
sub(X,Y,Z) :- add(Z,Y,X) .
```

```
?- sub(s(s(0)),s(0),N) .  
N = s(0) ;  
false.  
  
?- sub(N,M,s(0)) .  
N = s(M) ;  
false.
```

Ein weiteres Beispiel

Länge einer Liste in Haskell:

```
length [] = 0
length (x:xs) = length xs + 1
```

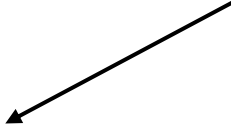
Länge einer Liste in Prolog:

```
length([],0).
length([X|Xs],N) :- length(Xs,M), N is M+1.
```

```
?- length([1,2,a],Res).
   Res = 3.
```

```
?- length(Liste,3).
   Liste = [_G331, _G334, _G337]
```

Liste mit 3 beliebigen
(variablen) Elementen



Arithmetik vs. symbolische Operatorterme

Vorsicht: wenn statt:

```
length([],0).  
length([X|Xs],N) :- length(Xs,M), N is M+1.
```

Verwendung von:

```
length([],0).  
length([X|Xs],M+1) :- length(Xs,M).
```

dann:

```
?- length([1,2,a],Res).  
Res = 0+1+1+1.
```

```
?- length(Liste,3).  
false.
```

```
?- length(Liste,0+1+1+1).  
Liste = [_G331, _G334, _G337].
```


Ein Beispiel entsprechend mehrerer verschachtelter Aufrufe

```
partition :: Int → [Int] → ([Int], [Int])
```

...

```
quicksort [ ] = [ ]  
quicksort (h : t) = quicksort l1 ++ h : quicksort l2  
  where (l1, l2) = partition h t
```



```
quicksort [ ] = [ ]  
quicksort (h : t) = ls ++ h : quicksort l2  
  where (l1, l2) = partition h t  
        ls = quicksort l1
```



```
quicksort [ ] = [ ]  
quicksort (h : t) = ls ++ h : lg  
  where (l1, l2) = partition h t  
        ls = quicksort l1  
        lg = quicksort l2
```



```
quicksort [ ] = [ ]  
quicksort (h : t) = list  
  where (l1, l2) = partition h t  
        ls = quicksort l1  
        lg = quicksort l2  
        list = ls ++ h : lg
```



```
quicksort([], []).  
quicksort([H|T], List) :-  
  partition(H, T, L1, L2),  
  quicksort(L1, LS),  
  quicksort(L2, LG),  
  append(LS, [H|LG], List).
```

Deskriptive Programmierung

Deklarative Semantik von Prolog

Was ist denn die „mathematische“ Bedeutung/Semantik eines Prolog-Programms?

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .
```

Logische Interpretation:

$$(\forall X. \text{add}(0, X, X)) \\ \wedge (\forall X, Y, Z. \text{add}(X, Y, Z) \Rightarrow \text{add}(s(X), Y, s(Z)))$$

Um solchen Formeln eine Bedeutung zuzuordnen, verwendet man in der Logik Modelle:

- Menge von Objekten
- Interpretation von Funktoren (wie „s(...)“) als Funktionen auf Objekten
- Interpretation von Prädikaten (wie „add(...)“) als Relationen zwischen Objekten
- Zuweisung von Wahrheitswerten zu Formeln nach festen Regeln
- Betrachtung nur von Interpretationen, die **alle gegebenen** Formeln wahr machen

Semantik eines Programms wäre gegeben durch alle Zusammenhänge, die in **allen** Modellen des Programms gelten.

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .
```

$$(\forall X. \text{add}(0, X, X))$$
$$\wedge (\forall X, Y, Z. \text{add}(X, Y, Z) \Rightarrow \text{add}(s(X), Y, s(Z)))$$

- Modell 1: Objekte: natürliche Zahlen
 Interpretation von 0 als 0
 Interpretation von s(...) als $s(n) = n + 1$
 Interpretation von add(...) als $\text{add}(n, m, k)$ gdw. $n + m = k$
- Modell 2: Objekte: $\{*\}$
 Interpretation von 0 als $*$
 Interpretation von s(...) als $s(*) = *$
 Interpretation von add(...) als $\text{add}(*, *, *)$ wahr
- Modell 3: Objekte: nichtpositive ganze Zahlen
 Interpretation von 0 als 0
 Interpretation von s(...) als $s(n) = n - 1$
 Interpretation von add(...) als $\text{add}(n, m, k)$ gdw. $n + m = k$

Herbrand-Modelle

Wichtig: Es gibt stets eine Art „Universalmodell“.

Idee: Interpretation möglichst einfach, nämlich rein syntaktisch.
Weder Funktoren noch Prädikate „tun“ irgendwas. **das Herbrand-Universum**

Also: Menge von Objekten = alle Grundterme (über gegebener Signatur)
Interpretation von Funktoren = syntaktische Anwendung auf Terme
Interpretation von Prädikaten = irgendeine Menge von Anwendungen von Prädikatssymbolen auf Grundterme

eine Herbrand-Interpretation

Beispiel:

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .
```

Herbrand-Universum: $\{0, s(0), s(s(0)), s(s(s(0))), \dots\}$ (ohne Prädikatssymbole!)

die Herbrand-Basis: $\{\text{add}(0, 0, 0), \text{add}(0, 0, s(0)), \text{add}(0, s(0), 0), \dots\}$

(**alle** Anwendungen von Prädikatssymbolen auf Terme des Herbrand-Universums)

Eine Herbrand-Interpretation ist **irgendeine Teilmenge** der Herbrand-Basis.

Beispiel:

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .
```

Herbrand-Interpretation 1: $\{\text{add}(0, 0, 0), \text{add}(0, 0, s(0)), \text{add}(0, s(0), 0), \dots\}$

Herbrand-Interpretation 2: \emptyset

Herbrand-Interpretation 3: $\{\text{add}(0, 0, 0), \text{add}(0, s(0), s(0)),$
 $\text{add}(s(0), 0, s(0)), \text{add}(s(0), s(0), s(s(0))), \dots\}$

Unser Ziel ist eine Herbrand-Interpretation, die alle durch das Programm gegebenen Formeln wahr macht, aber auch nicht unnötig mehr wahr macht.

Herbrand-Modelle

Eine Herbrand-Interpretation ist ein Modell für ein Programm, wenn für jede vollständige Instanziierung (**keine Variablen übrig**)

$$L_0 :- L_1, L_2, \dots, L_n$$

jeder Klausel gilt: wenn L_1, L_2, \dots, L_n in der Interpretation, dann auch L_0 .

Beispiel:

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .
```

$$(\forall X. \text{add}(0, X, X))$$
$$\wedge (\forall X, Y, Z. \text{add}(X, Y, Z) \Rightarrow \text{add}(s(X), Y, s(Z)))$$

- Die Herbrand-Basis ist (immer) ein Modell.
- Die Herbrand-Interpretation $\emptyset = \{\}$ ist (hier) kein Modell.
- Die Interpretation $\{\text{add}(0, 0, 0), \text{add}(0, s(0), s(0)), \text{add}(s(0), 0, s(0)), \text{add}(s(0), s(0), s(s(0))), \dots\}$ ist hier ein Modell.

Kleinstes Herbrand-Modell

Die deklarative Bedeutung eines Programms ist seine **kleinste Herbrand-Interpretation, die ein Modell ist!**

Für das Beispiel:

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .
```

$\{ \text{add}(0, 0, 0), \text{add}(0, s(0), s(0)), \text{add}(s(0), 0, s(0)), \text{add}(s(0), s(0), s(s(0))), \dots \}$

Allgemein:

Gibt es immer so ein kleinstes Modell?

Ja, weil (Herbrand)-Modelle, für Programme bestehend aus sogenannten Horn-Klauseln (genau die in Prolog ohne Negation vorkommenden), unter Durchschnitt abgeschlossen sind!

Kann man das kleinste Herbrand-Modell (mathematisch konstruktiv) „ausrechnen“?

Ja, mittels des „Immediate Consequence Operators“: T_P

Definition: T_P nimmt eine Interpretation I und erzeugt alle Grundlitterale (Elemente der Herbrand-Basis) L_0 , für die L_1, L_2, \dots, L_n in I existieren, so dass $L_0 :- L_1, L_2, \dots, L_n$ eine vollständige Instanziierung irgendeiner der gegebenen Programm-Klauseln ist.

Offenbar: Eine Herbrand-Interpretation I ist ein Modell gdw. $T_P(I)$ Teilmenge von I .

Außerdem: Das kleinste Herbrand-Modell ergibt sich als Fixpunkt/Limit der Folge

$$\emptyset, T_P(\emptyset), T_P(T_P(\emptyset)), T_P(T_P(T_P(\emptyset))), \dots$$

Am Beispiel:

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .
```

$$T_P(\emptyset) = \{\text{add}(0, 0, 0), \text{add}(0, s(0), s(0)), \text{add}(0, s(s(0)), s(s(0))), \dots\}$$

$$T_P(T_P(\emptyset)) = T_P(\emptyset) \cup \{\text{add}(s(0), 0, s(0)), \text{add}(s(0), s(0), s(s(0))), \\ \text{add}(s(0), s(s(0)), s(s(s(0))))\}, \dots\}$$

$$T_P(T_P(T_P(\emptyset))) = T_P(T_P(\emptyset)) \cup \{\text{add}(s(s(0)), 0, s(s(0))), \\ \text{add}(s(s(0)), s(0), s(s(s(0))))\}, \\ \text{add}(s(s(0)), s(s(0)), s(s(s(s(0))))\}, \dots\}$$

...

Für welche Art von Programmen kann man mit der T_P -Semantik arbeiten?

- keine Arithmetik, kein **is**
- kein $\backslash=$, kein **not**
- allgemein, keine der (noch einzuführenden) „nicht-logischen“ Features

Aber eben zum Beispiel Programme wie:

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .  
  
mult(0, _, 0) .  
mult(s(_), 0, 0) .  
mult(s(X), s(Y), s(Z)) :- mult(X, s(Y), U), add(Y, U, Z) .
```

$$T_P(\emptyset) = \{\text{add}(0, 0, 0), \text{add}(0, s(0), s(0)), \dots\} \cup \{\text{mult}(0, 0, 0), \text{mult}(0, s(0), 0), \dots\} \cup \{\text{mult}(s(0), 0, 0), \dots\}$$

$$T_P(T_P(\emptyset)) = T_P(\emptyset) \cup \{\text{add}(s(0), 0, s(0)), \text{add}(s(0), s(0), s(s(0))), \dots\} \cup \{\text{mult}(s(0), s(0), s(0))\}$$

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .  
  
mult(0, _, 0) .  
mult(s(_), 0, 0) .  
mult(s(X), s(Y), s(Z)) :- mult(X, s(Y), U), add(Y, U, Z) .
```

$$T_P(\emptyset) = \{\text{add}(0, 0, 0), \text{add}(0, s(0), s(0)), \dots\} \cup \{\text{mult}(0, 0, 0), \text{mult}(0, s(0), 0), \dots\} \cup \{\text{mult}(s(0), 0, 0), \dots\}$$

$$T_P(T_P(\emptyset)) = T_P(\emptyset) \cup \{\text{add}(s(0), 0, s(0)), \text{add}(s(0), s(0), s(s(0))), \dots\} \cup \{\text{mult}(s(0), s(0), s(0))\}$$

$$T_P(T_P(T_P(\emptyset))) = T_P(T_P(\emptyset)) \cup \{\text{add}(s(s(0)), 0, s(s(0))), \dots\} \cup \{\text{mult}(s(0), s(s(0)), s(s(0))), \text{mult}(s(s(0)), s(0), s(s(0)))\}$$

$$T_P^4(\emptyset) = T_P^3(\emptyset) \cup \{\text{add}(s^3(0), 0, s^3(0)), \text{add}(s^3(0), s(0), s^4(0)), \dots\} \cup \{\text{mult}(s(0), s^3(0), s^3(0)), \text{mult}(s^2(0), s^2(0), s^4(0)), \text{mult}(s^3(0), s(0), s^3(0))\}$$

Die deklarative Semantik:

- ist nur auf bestimmte, „rein logische“, Programme anwendbar
- beschreibt nicht direkt das Verhalten bei Anfragen mit Variablen
- ist mathematisch einfacher als die noch einzuführende operationelle Semantik
- lässt sich geeignet zur operationellen Semantik in Beziehung setzen
- ist insensitiv gegenüber Änderungen der Reihenfolge von und innerhalb Regeln (!)

Operationalisierung?

Spezifikation („Programm“) \equiv
Relationsdefinitionen

```
istVaterVon(kurt,fritz).  
istVaterVon(fritz,paul).  
istVaterVon(fritz,hans).  
  
istGrossvaterVon(G,E) :-  
    istVaterVon(G,V),istVaterVon(V,E).  
istGrossvaterVon(G,E) :-  
    istVaterVon(G,M),istMutterVon(M,E).
```

?- istGrossvaterVon(kurt,X)

~> ...

~> ...

~> ...

~> ...

~> **X = paul ; X = hans**

Eingabe: eine Anfrage



(wiederholte) Reduktion



Ausgabe: Variablensubstitution(en)

Operationalisierung in Prolog (1)

Operationalisierungsprinzip: Rückführung auf ein Unterproblem (**Reduktion**)

`istGrossvaterVon(kurt, X)`

Matching/
Parameter-
übergabe

```
istVaterVon(kurt, fritz) .  
istVaterVon(fritz, paul) .  
istVaterVon(fritz, hans) .
```

```
istGrossvaterVon(G,E) :- istVaterVon(G,V), istVaterVon(V,E) .  
istGrossvaterVon(G,E) :- istVaterVon(G,M), istMutterVon(M,E) .
```

1te Reduktion

`istVaterVon(kurt, V)`

Operationalisierung in Prolog (2)

Operationalisierungsprinzip: Rückführung auf ein Unterproblem, wobei neue Unteranfragen von links nach rechts gefunden werden!

istGrossvaterVon(kurt, X)

Matching/
Parameter-
übergabe

```
istVaterVon(kurt, fritz) .  
istVaterVon(fritz, paul) .  
istVaterVon(fritz, hans) .
```

```
istGrossvaterVon(G,E) :- istVaterVon(G,V), istVaterVon(V,E) .  
istGrossvaterVon(G,E) :- istVaterVon(G,M), istMutterVon(M,E) .
```

istVaterVon(kurt, fritz)

istVaterVon(fritz, E)

2te Reduktion

Operationalisierung in Prolog (3)

Operationalisierungsprinzip: Rückführung auf ein Unterproblem (**Reduktion**)

istGrossvaterVon (kurt, X)

Matching/
Parameter-
übergabe

Rückgabe der
Ergebnis-
parameter

```
istVaterVon (kurt, fritz) .  
istVaterVon (fritz, paul) .  
istVaterVon (fritz, hans) .
```

```
istGrossvaterVon (G,E) :- istVaterVon (G,V) , istVaterVon (V,E) .  
istGrossvaterVon (G,E) :- istVaterVon (G,M) , istMutterVon (M,E) .
```

istVaterVon (kurt, fritz)

E = paul

istVaterVon (fritz, paul)

Operationalisierung in Prolog (4)

- Prolog sucht nach matchenden Regeln oder Fakten immer von oben nach unten im Programm.

Unteranfrage: `istVaterVon(fritz,E)`

```
istVaterVon(kurt,fritz).  
istVaterVon(fritz,paul).  
istVaterVon(fritz,hans).
```

Lösung:

E = paul

- Da eine Relation keine eindeutige Abbildung ist, können weitere Antworten für eine (Unter-)Anfrage existieren. Prolog findet diese mittels **Backtracking**:

Reevaluierung: `istVaterVon(fritz,E)`

Position letzter
Lösung – da wird
weitergesucht

```
istVaterVon(kurt,fritz).  
istVaterVon(fritz,paul).  
istVaterVon(fritz,hans).
```

Lösung:

E = paul ;

E = hans

Operationalisierung in Prolog (5)

Operationalisierungsprinzip: Rückführung auf ein Unterproblem (**Reduktion**)

istGrossvaterVon(kurt, X)

Matching/
Parameter-
übergabe

Rückgabe der
Ergebnis-
parameter

```
istVaterVon(kurt, fritz) .  
istVaterVon(fritz, paul) .  
istVaterVon(fritz, hans) .
```

```
istGrossvaterVon(G, E) :- istVaterVon(G, V), istVaterVon(V, E) .  
istGrossvaterVon(G, E) :- istVaterVon(G, M), istMutterVon(M, E) .
```

istVaterVon(kurt, fritz)

E = hans

istVaterVon(fritz, hans)

Operationalisierung in Prolog (6)

Das **Backtracking** bezieht sich auch auf andere „matchende“ Regeln:

istGrossvaterVon (kurt, X)

Matching/
Parameter-
übergabe

```
istVaterVon (kurt, fritz) .  
istVaterVon (fritz, paul) .  
istVaterVon (fritz, hans) .
```

```
istGrossvaterVon (G,E) :- istVaterVon (G,V) , istVaterVon (V,E) .  
istGrossvaterVon (G,E) :- istVaterVon (G,M) , istMutterVon (M,E) .
```

3te Reduktion

istVaterVon (kurt, M)

Fehlschlag !

istMutterVon (fritz, E)

Operationalisierung am Beispiel nochmal anders dargestellt

```
istVaterVon(kurt, fritz) .  
istVaterVon(fritz, paul) .  
istVaterVon(fritz, hans) .  
  
istGrossvaterVon(G, E) :-  
    istVaterVon(G, V), istVaterVon(V, E) .  
istGrossvaterVon(G, E) :-  
    istVaterVon(G, M), istMutterVon(M, E) .
```

X = paul:

```
?- istGrossvaterVon(kurt, X).  
?- istVaterVon(kurt, V), istVaterVon(V, X).  
?- istVaterVon(fritz, X).  
?- .
```

Vergleiche (innerhalb eines Prolog-Systems):
Benutzung von ?- trace.

Operationalisierung am Beispiel nochmal anders dargestellt

```
istVaterVon(kurt, fritz) .  
istVaterVon(fritz, paul) .  
istVaterVon(fritz, hans) .  
  
istGrossvaterVon(G, E) :-  
    istVaterVon(G, V), istVaterVon(V, E) .  
istGrossvaterVon(G, E) :-  
    istVaterVon(G, M), istMutterVon(M, E) .
```

X = paul:

X = hans:

```
?- istGrossvaterVon(kurt, X).  
?- istVaterVon(kurt, V), istVaterVon(V, X).  
?- istVaterVon(fritz, X).  
?- .  
?- .
```

Vergleiche (innerhalb eines Prolog-Systems):
Benutzung von ?- trace.

Operationalisierung am Beispiel nochmal anders dargestellt

```
istVaterVon(kurt, fritz) .
istVaterVon(fritz, paul) .
istVaterVon(fritz, hans) .

istGrossvaterVon(G, E) :-
    istVaterVon(G, V), istVaterVon(V, E) .
istGrossvaterVon(G, E) :-
    istVaterVon(G, M), istMutterVon(M, E) .
```

X = paul:

X = hans:

```
?- istGrossvaterVon(kurt, X).
?- istVaterVon(kurt, V), istVaterVon(V, X).
?- istVaterVon(fritz, X).
?- .
?- .
?- istVaterVon(kurt, M), istMutterVon(M, X).
?- istMutterVon(fritz, X).
```

Fehlschlag !

Vergleiche (innerhalb eines Prolog-Systems):
Benutzung von ?- trace.

Deskriptive Programmierung

Operationelle Semantik von Prolog

Motivation: Beobachtung einiger nicht so schöner (nicht so „logischer“?) Effekte

```
direct(frankfurt,san_francisco).
direct(frankfurt,chicago).
direct(san_francisco,honolulu).
direct(honolulu,maui).

connection(X, Y) :- direct(X, Y).
connection(X, Y) :- direct(X, Z), connection(Z, Y).
```

```
?- connection(frankfurt,maui).
true.

?- connection(san_francisco,X).
X = honolulu ;
X = maui ;
false.

?- connection(maui,X).
false.
```

Motivation: Beobachtung einiger nicht so schöner (nicht so „logischer“?) Effekte

```
direct(frankfurt,san_francisco).
direct(frankfurt,chicago).
direct(san_francisco,honolulu).
direct(honolulu,maui).

connection(X, Y) :- connection(X, Z), direct(Z, Y).
connection(X, Y) :- direct(X, Y).
```

```
?- connection(frankfurt,maui).
ERROR: Out of local stack
```

- Offenbar sind die impliziten logischen Operationen nicht kommutativ.
- Hinter der Programmausführung steckt also mehr als die rein logische Lesart.

Etwas subtiler...

```
add(0,X,X) .  
add(s(X),Y,s(Z)) :- add(X,Y,Z) .  
  
sub(X,Y,Z) :- add(Z,Y,X) .
```

```
?- sub(N,M,s(0)) .  
N = s(M) ;  
false.
```



```
add(X,0,X) .  
add(X,s(Y),s(Z)) :- add(X,Y,Z) .  
  
sub(X,Y,Z) :- add(Z,Y,X) .
```

```
?- sub(s(s(0)),s(0),N) .  
N = s(0) ;  
false.  
  
?- sub(N,M,s(0)) .  
N = s(0) ,  
M = 0 ;  
N = s(s(0)) ,  
M = s(0) ;
```

Die Wahl/Behandlung der Reihenfolge von Argumenten in Definitionen beeinflusst also die Qualität der Ergebnisse.

...

... und (daher) manchmal weniger Flexibilität als gewünscht

Die schön deskriptive Lösung:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```

funktioniert sehr gut für eine Reihe von Anfragemustern:

```
?- mult(s(s(0)),s(s(s(0))),N).
N = s(s(s(s(s(0))))).

?- mult(s(s(0)),N,s(s(s(s(0))))).
N = s(s(0)) ;
false.
```

Man sagt, `mult` unterstützt die „Aufrufmodi“ `mult(+X,+Y,?Z)` und `mult(+X,?Y,+Z)`

Aber es gibt auch „Ausreißer“:

```
?- mult(N,M,s(s(s(s(0))))).
N = s(0),
M = s(s(s(s(0)))) ;
N = s(s(0)),
M = s(s(0)) ;
abort
```

... aber nicht `mult(?X,?Y,+Z)`.

sonst Endlossuche

... und (daher) manchmal weniger Flexibilität als gewünscht

Hingegen bei nur der Addition:

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .
```

hatte das ja gut geklappt:

```
?- add(N, M, s(s(s(0)))) .  
N = 0,  
M = s(s(s(0))) ;  
N = s(0),  
M = s(s(0)) ;  
N = s(s(0)),  
M = s(0) ;  
N = s(s(s(0))),  
M = 0 ;  
false.
```

In der Tat unterstützt **add** alle Aufrufmodi, sogar **add(?X, ?Y, ?Z)**.

1. Warum der Unterschied?
2. Was kann man tun, um **mult** auch so funktionieren zu lassen?

Außerdem Vorsicht bei Verwendung/Positionierung negativer Aussagen nötig

Und, nun wird es ganz „komisch“:

```
loves (vincent, mia) .  
loves (marsellus, mia) .  
loves (mia, vincent) .  
  
jealous (X, Y) :- loves (X, Z) , loves (Y, Z) , X \= Y.
```



kleine Änderung

```
...  
  
jealous (X, Y) :- X \= Y, loves (X, Z) , loves (Y, Z) .
```

```
?- jealous (marsellus, X) .  
false.  
  
?- jealous (X, _) .  
false.  
  
?- jealous (X, Y) .  
false.
```

Hingegen hatten wir vor der kleinen Änderung hier jeweils sinnvolle Ergebnisse gekriegt!

Um all diesen Phänomenen nachzugehen, müssen wir uns mit dem konkreten Prolog-Ausführungsmechanismus beschäftigen.

„Zutaten“ für diese Diskussion der operationellen Semantik, im folgenden betrachtet:

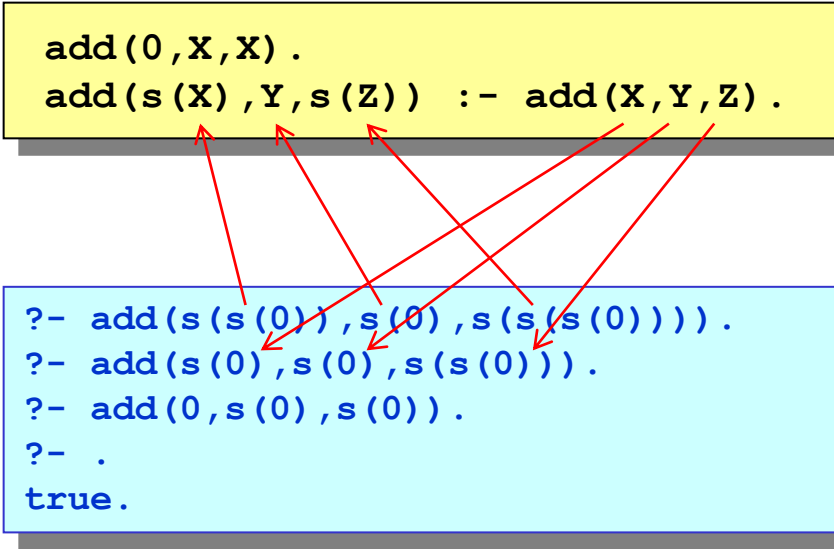
1. Unifikation
2. Resolution
3. Ableitungsbäume

Deskriptive Programmierung

Unifikation

Analogie zu Haskell: Pattern Matching

```
add(0,X,X) .  
add(s(X),Y,s(Z)) :- add(X,Y,Z) .
```



```
?- add(s(s(0)),s(0),s(s(0))) .  
?- add(s(0),s(0),s(s(0))) .  
?- add(0,s(0),s(0)) .  
?- .  
true.
```

Aber was ist mit „Ausgabevariablen“?

```
add(0,X,X) .  
add(s(X),Y,s(Z)) :- add(X,Y,Z) .
```

?



```
?- add(s(s(0)),s(0),N) .
```

Gleichheit „ $=$ “ als zweistelliges Prolog-Prädikat, das eine Menge leistet:

- Vergleiche auf Grundtermen (Terme ohne Variablen) durchführen, zum Beispiel:

$$\begin{aligned} s(0) = s(0) &\Rightarrow \text{true} \\ s(0) = s(s(0)) &\Rightarrow \text{false} \end{aligned}$$

- Bindung von Variablen akzeptieren, zum Beispiel:

$$\begin{aligned} N=0 &\Rightarrow \text{true} \\ N=s(U) &\Rightarrow \text{true} \\ s(0)=N &\Rightarrow \text{true} \\ M=V &\Rightarrow \text{true} \end{aligned}$$

- strukturell matchen und binden, zum Beispiel:

$$\begin{aligned} s(s(0)) = s(V) &\Rightarrow V=s(0) \\ s(U) = s(0) &\Rightarrow U=0 \end{aligned}$$

- Bindungen „aufsammeln“/verknüpfen, zum Beispiel:

$$N=s(V), M=V \Rightarrow N=s(M)$$

Gleichheit von Termen (1)

- Überprüfung der Gleichheit von Grundtermen:

<code>europa = europa ?</code>	<code>yes</code>
<code>person(fritz,mueller) = person(fritz,mueller) ?</code>	<code>yes</code>
<code>person(fritz,mueller) = person(mueller,fritz) ?</code>	<code>no</code>
<code>5 = 2 ?</code>	<code>no</code>
<code>5 = 2 + 3 ?</code>	<code>no</code>
<code>2 + 3 = +(2, 3) ?</code>	<code>yes</code>

⇒ Gleichheit von Termen bedeutet **strukturelle** Gleichheit.

Terme werden vor einem Vergleich nicht „ausgewertet“!

Gleichheit von Termen (2)

- Überprüfung der Gleichheit von Termen mit Variablen:

```
person(fritz, Nachname, datum(27, 11, 2007))  
= person(fritz, mueller, datum(27, MM, 2007)) ?
```

- Für eine Variable darf jeder beliebige Term eingesetzt werden:
 - insbesondere **mueller** für **Nachname** und **11** für **MM**
 - Nach dieser Ersetzung sind beide Terme gleich.

Gleichheit von Termen (3)

Welche Variablen müssen wie ersetzt werden, um die Terme anzugleichen?

```
datum(1, 4, 1985) = datum(1, 4, Jahr) ?  
datum(Tag, Monat, 1985) = datum(1, 4, Jahr) ?  
a(b, C, d(e, F, g(h, i, J))) = a(B, c, d(E, f, g(H, i, K))) ?  
X = Y + 1 ?  
[[the, Y]|Z] = [[X, dog], [is, here]] ?
```

Zur Erinnerung, Listensyntax:

```
[1,2,a] = [1|[2,a]] = [1,2|[a]] = [1,2|. (a, [])] = . (1, . (2, . (a, [])))
```

Und was ist mit:

```
p(X) = p(q(X)) ?
```

„occurs check“ (siehe später)

Gleichheit von Termen (4)

Einige weitere (problematische) Fälle:

```
loves(vincent, X) = loves(X, mia) ?
```

```
loves(marcellus, mia) = loves(X, X) ?
```

```
a(b, C, d(e, F, g(h, i, J))) = a(B, c, d(E, f, p(H, i, K))) ?
```

```
p(b, b) = p(X) ?
```

```
...
```

Substitution:

- Ersetzen von Variablen durch andere Variablen oder andere Formen von Termen (Konstanten, Strukturen, ...)
- Abbildung, die jedem Term eindeutig einen neuen Term zuordnet, wobei sich der neue vom alten Term nur durch die Ersetzung von Variablen unterscheidet.

- Notation:

$$U = \{\text{Nachname} / \text{mueller}, \text{MM} / 11\}$$

- Die Substitution U verändert nur die Variablen **Nachname** und **MM**, alles andere bleibt unverändert!
- $U(\text{person}(\text{fritz}, \text{Nachname}, \text{datum}(27, 11, 2007)))$
 $\quad == \text{person}(\text{fritz}, \text{mueller}, \text{datum}(27, 11, 2007))$

- Unifikator:

- Substitution, die zwei Terme „gleichmacht“.
- z.B., Anwendung der Substitution $U = \{ \text{Nachname}/\text{mueller}, \text{MM}/11 \}$:

$$\begin{aligned} &U(\text{person}(\text{fritz}, \text{Nachname}, \text{datum}(27, 11, 2007))) \\ &== U(\text{person}(\text{fritz}, \text{mueller}, \text{datum}(27, \text{MM}, 2007))) \end{aligned}$$

- allgemeinster Unifikator:

- Unifikator, der möglichst viele Variablen unverändert lässt.
- Beispiel: $\text{datum}(\text{TT}, \text{MM}, 2007)$ und $\text{datum}(\text{T}, 11, \text{J})$

- $U_1 = \{ \text{TT}/27, \text{T}/27, \text{MM}/11, \text{J}/2007 \}$ 

- $U_2 = \{ \text{TT}/\text{T}, \text{MM}/11, \text{J}/2007 \}$ 


- Prolog sucht immer einen allgemeinsten Unifikator.

Unifikation, formal (3) - Berechnung eines allgemeinsten Unifikators

Eingabe: zwei Terme T_1 und T_2 (im allgemeinen mit ggfs. gemeinsamen Variablen)

Ausgabe: ein allgemeinsten Unifikator U für T_1 und T_2 ,
falls T_1 und T_2 unifizierbar sind, ansonsten Fehlschlag

Methode:

1. Wenn T_1 und T_2 gleiche Konstanten oder Variablen sind,
dann ist $U = \emptyset$
 2. Wenn T_1 eine Variable ist, die nicht in T_2 vorkommt,
dann ist $U = \{T_1 / T_2\}$
 3. Wenn T_2 eine Variable ist, die nicht in T_1 vorkommt,
dann ist $U = \{T_2 / T_1\}$
- „occurs check“
- 

Methode (Forts.):

4. Falls $T_1 = f(T_{1,1}, \dots, T_{1,n})$ und $T_2 = f(T_{2,1}, \dots, T_{2,n})$ Strukturen mit dem gleichen Funktor und der gleichen Anzahl von Komponenten sind, dann
 1. Finde einen allgemeinsten Unifikator U_1 für $T_{1,1}$ und $T_{2,1}$
 2. Finde einen allgemeinsten Unifikator U_2 für $U_1(T_{1,2})$ und $U_1(T_{2,2})$
 - ...
 - n. Finde einen allgemeinsten Unifikator U_n für
 $U_{n-1}(\dots(U_1(T_{1,n}))\dots)$ und $U_{n-1}(\dots(U_1(T_{2,n}))\dots)$

Falls alle diese Unifikatoren existieren, dann ist

$$U = U_n \circ U_{n-1} \circ \dots \circ U_1 \quad (\text{Komposition der Unifikatoren})$$

5. Sonst: T_1 und T_2 sind nicht unifizierbar.

$\text{datum}(1, 4, 1985) = \text{datum}(1, 4, \text{Jahr})$?

Strukturen mit gleichem Funktor, gleicher Anzahl von Komponenten, also:

1. Finde einen allgemeinsten Unifikator U_1 für **1** und **1**
 \Rightarrow gleiche Konstanten, daher $U_1 = \emptyset$
2. Finde einen allgemeinsten Unifikator U_2 für $U_1(\mathbf{4})$ und $U_1(\mathbf{4})$
 \Rightarrow gleiche Konstanten, daher $U_2 = \emptyset$
3. Finde einen allgemeinsten Unifikator U_3 für $U_2(U_1(\mathbf{1985}))$ und $U_2(U_1(\mathbf{Jahr}))$
 \Rightarrow Konstante vs. Variable, daher $U_3 = \{\mathbf{Jahr} / \mathbf{1985}\}$

Ein allgemeinsten Unifikator insgesamt ist:

$$U = U_3 \circ U_2 \circ U_1 = \{\mathbf{Jahr} / \mathbf{1985}\}$$

`loves(marcellus, mia) = loves(X, X) ?`

Strukturen mit gleichem Funktor, gleicher Anzahl von Komponenten, also:

1. Finde einen allgemeinsten Unifikator U_1 für `marcellus` und `X`
⇒ Konstante vs. Variable, daher $U_1 = \{X/marcellus\}$
2. Finde einen allgemeinsten Unifikator U_2 für $U_1(mia)$ und $U_1(X)$
⇒ **verschiedene** Konstanten, daher existiert U_2 nicht!

Folglich existiert auch kein Unifikator für die Ausgangsterme!

$$d(\mathbf{E}, g(\mathbf{H}, \mathbf{J})) = d(\mathbf{F}, g(\mathbf{H}, \mathbf{E})) \quad ?$$

Strukturen mit gleichem Funktor, gleicher Anzahl von Komponenten, also:

1. Finde einen allgemeinsten Unifikator U_1 für \mathbf{E} und \mathbf{F}
 \Rightarrow verschiedene Variablen, daher $U_1 = \{\mathbf{E}/\mathbf{F}\}$
2. Finde einen allgemeinsten Unifikator U_2 für $U_1(g(\mathbf{H}, \mathbf{J}))$ und $U_1(g(\mathbf{H}, \mathbf{E}))$

$$g(\mathbf{H}, \mathbf{J}) = g(\mathbf{H}, \mathbf{E}) \quad ?$$

\Rightarrow Strukturen mit gleichem Funktor, gleicher Anzahl von Komponenten, also:

- Finde einen allgemeinsten Unifikator $U_{2,1}$ für \mathbf{H} und \mathbf{H}

\Rightarrow gleiche Variablen, daher $U_{2,1} = \emptyset$

- Finde einen allgemeinsten Unifikator $U_{2,2}$ für $U_{2,1}(\mathbf{J})$ und $U_{2,1}(\mathbf{E})$

\Rightarrow verschiedene Variablen, daher $U_{2,2} = \{\mathbf{E}/\mathbf{J}\}$

$$U_2 = U_{2,2} \circ U_{2,1} = \{\mathbf{E}/\mathbf{J}\}$$

Ein allgemeinsten Unifikator insgesamt ist:

$$U = U_2 \circ U_1 = \{\mathbf{E}/\mathbf{J}, \mathbf{F}/\mathbf{J}\}$$

Zur Erinnerung:

2. Wenn T_1 eine Variable ist, die nicht in T_2 vorkommt, dann ist $U = \{T_1 / T_2\}$
 3. Wenn T_2 eine Variable ist, die nicht in T_1 vorkommt, dann ist $U = \{T_2 / T_1\}$
- „occurs check“

Also zum Beispiel:

$$\mathbf{x} = \mathbf{q}(\mathbf{x}) \quad ?$$

⇒ Es existiert kein Unifikator.

In Prolog wird diese Überprüfung jedoch standardmäßig nicht durchgeführt!

Ohne „occurs check“:

$$p(\mathbf{x}) = p(\mathbf{q}(\mathbf{x})) ?$$

Strukturen mit gleichem Funktor, gleicher Anzahl von Komponenten, also:

1. Finde einen allgemeinsten Unifikator U_I für \mathbf{x} und $\mathbf{q}(\mathbf{x})$
 \Rightarrow Variable vs. Term, daher $U_I = \{\mathbf{x}/\mathbf{q}(\mathbf{x})\}$

$$U = U_I = \{\mathbf{x}/\mathbf{q}(\mathbf{x})\} !$$

Obwohl es ja eigentlich nicht stimmt, dass $U(p(\mathbf{x}))$ und $U(p(\mathbf{q}(\mathbf{x})))$ gleich sind!

Deskriptive Programmierung

Resolution

Resolutions-/(Beweis)prinzip in Prolog

Man kann den Beweis der Anfrage

$?- P, L, Q.$

(P, L und Q seien **variablenfreie** Literale)

auf den Beweis der Anfrage

$?- P, L_1, L_2, \dots, L_n, Q.$

zurückführen, wenn $L :- L_1, L_2, \dots, L_n.$ eine Regel ist (wobei $n \geq 0$).

- Die Wahl des Literals L und der Regel dafür sind prinzipiell beliebig.
- Ist $n = 0$, so wird die Anfrage durch den Resolutionsschritt kürzer.

Resolutionsprinzip – mit Variablen

Man kann den Beweis der Anfrage

$?- P, L, Q.$

(P, L und Q sind Literale)

auf den Beweis der Anfrage

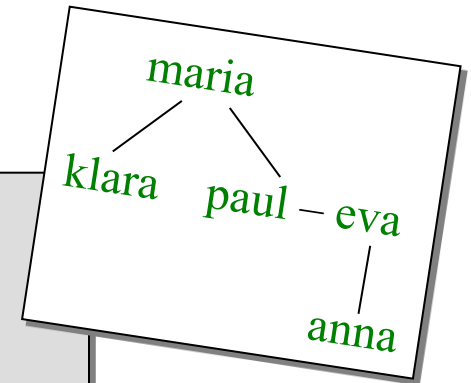
$?- U(P), U(L_1), U(L_2), \dots, U(L_n), U(Q).$

zurückführen, wenn

- es eine Regel $L_0 :- L_1, L_2, \dots, L_n.$ gibt ($n \geq 0$), mit „zur Sicherheit“ umbenannten Variablen (so dass keine Überschneidung mit denen in P, L, Q), und
- U ein **allgemeinster Unifikator** von L und L_0 ist.

Resolution in Prolog (3)

```
istMutterVon(maria, klara).  
istMutterVon(maria, paul).  
istMutterVon(eva, anna).  
  
istVerheiratetMit(paul, eva).  
  
istGrossmutterVon(G, E) :-  
    istMutterVon(G, M), istMutterVon(M, E).  
istGrossmutterVon(G, E) :-  
    istMutterVon(G, V), istVaterVon(V, E).  
  
istVaterVon(V, K) :-  
    istVerheiratetMit(V, M), istMutterVon(M, K).  
  
...
```



Resolution in Prolog (4)

?- istGrossmutterVon(maria,anna).

```
istGrossmutterVon(G1, E1) :-  
    istMutterVon(G1, V1), istVaterVon(V1, E1).  
⇒  $U_1 = \{G1/ maria, E1/ anna\}$ 
```

-> istMutterVon(maria,V1), istVaterVon(V1,anna).

```
istMutterVon(maria, paul).  
⇒  $U_2 = \{V1/ paul\}$ 
```

-> istVaterVon(paul,anna).

```
istVaterVon(V2, K2) :-  
    istVerheiratetMit(V2, M2), istMutterVon(M2, K2).  
⇒  $U_3 = \{V2/ paul, K2/ anna\}$ 
```

-> istVerheiratetMit(paul,M2), istMutterVon(M2,anna).

```
istVerheiratetMit(paul, eva).  
⇒  $U_4 = \{M2/ eva\}$ 
```

-> istMutterVon(eva,anna).

```
istMutterVon(eva, anna).  
⇒  $U_5 = \emptyset$ 
```

-> □

Resolution in Prolog (5)

```
?- istGrossmutterVon(maria,anna).
```

$$U_1 = \{G1/ maria, E1/anna\}$$

$$U_2 = \{V1/paul\}$$

$$U_3 = \{V2/paul, K2/anna\}$$

$$U_4 = \{M2/eva\}$$

$$U_5 = \emptyset$$

Die Antwort auf die Anfrage ist die Substitution \mathbf{U} aller in der Anfrage vorkommenden Variablen, die die Variablen genauso ersetzt wie die Substitution

$$U_5 \circ U_4 \circ U_3 \circ U_2 \circ U_1 = \{G1/ maria, E1/anna, V1/paul, V2/paul, K2/anna, M2/eva\}$$

Hier: $\mathbf{U} = \emptyset$.

Deskriptive Programmierung

Ableitungsbäume

Zur Erinnerung, Motivation für Betrachtung operationeller Semantik ...

Wir wollten zum Beispiel verstehen, warum für

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .  
  
mult(0, _, 0) .  
mult(s(X), Y, Z) :- mult(X, Y, U), add(U, Y, Z) .
```

eine Reihe von Anfragemustern/„Aufrufmodi“ sehr gut funktioniert:

```
?- mult(s(s(0)), s(s(s(0))), N) .  
N = s(s(s(s(s(s(0)))))) .  
  
?- mult(s(s(0)), N, s(s(s(s(0)))) .  
N = s(s(0)) ;  
false.
```

aber andere nicht:

```
?- mult(N, M, s(s(s(s(0)))) .  
N = s(0) ,  
M = s(s(s(s(0)))) ;  
N = s(s(0)) ,  
M = s(s(0)) ;  
abort
```

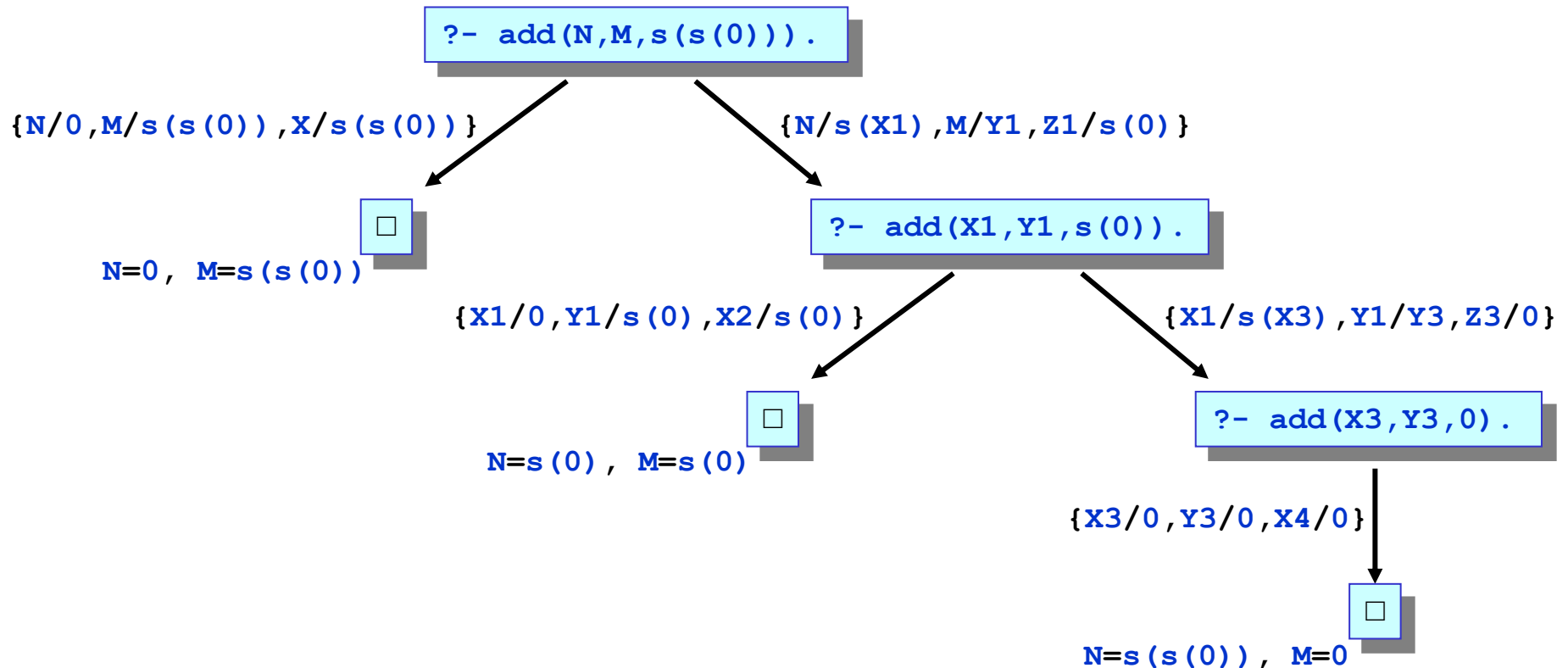
sonst Endlossuche

Explizite Aufzählung von Lösungen

Beginnen wir mit einem einfachen Beispiel für nur die Addition:

```
add(0, X, X).  
add(s(X), Y, s(Z)) :- add(X, Y, Z).
```

Vollständige Suche:



Ein Beispiel mit endloser Suche

```
add(0,X,X).  
add(s(X),Y,s(Z)) :- add(X,Y,Z).  
  
mult(0,_,0).  
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```

```
?- mult(N,M,s(0)).
```

{N/s(X),M/Y,Z/s(0)}

```
?- mult(X,Y,U),add(U,Y,s(0)).
```

{X/0,Y/_1,U/0}

{X/s(X2),Y/Y2,U/Z2}

```
?- add(0,_1,s(0)).
```

```
?- mult(X2,Y2,U2),add(U2,Y2,Z2),add(Z2,Y2,s(0)).
```

{_1/s(0),X1/s(0)}

{X2/0,Y2/_2,U2/0}

{X2/s(X3),Y2/Y3,U2/Z3}

N=s(0), M=s(0)

```
?- add(0,_2,Z2),add(Z2,_2,s(0)).
```

```
?- ...
```

Wird immer länger!

Probekalber Umordnung von Literalen

```
add(0,X,X).  
add(s(X),Y,s(Z)) :- add(X,Y,Z).  
  
mult(0,_,0).  
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```



```
add(0,X,X).  
add(s(X),Y,s(Z)) :- add(X,Y,Z).  
  
mult(0,_,0).  
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```

```
?- mult(N,M,s(0)).
```

{N/s(X),M/Y,Z/s(0)}

```
?- add(U,Y,s(0)),mult(X,Y,U).
```

{U/0,Y/s(0),X1/s(0)}

```
?- mult(X,s(0),0).
```

Probekalber Umordnung von Literalen

```

add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
    
```

```
?- mult(N,M,s(0)).
```

```
{N/s(X),M/Y,Z/s(0)}
```

```
?- add(U,Y,s(0)),mult(X,Y,U).
```

```
{U/0,Y/s(0),X1/s(0)}
```

```
{U/s(X3),Y/Y3,Z3/0}
```

```
?- mult(X,s(0),0).
```

```
?- add(X3,Y3,0),mult(X,Y3,s(X3)).
```

```
{X/0,_1/s(0)}
```

```
{X/s(X2),Y2/s(0),Z2/0}
```

```
{X3/0,Y3/0,X4/0}
```

□

```
?- add(U2,s(0),0),mult(X2,s(0),U2).
```

```
?- mult(X,0,s(0)).
```

```
N=s(0),
M=s(0)
```



```
{X/s(X5),Y5/0,Z5/s(0)}
```

```
?- add(U5,0,s(0)),mult(X5,0,U5).
```

Probekalber Umordnung von Literalen

```
add(0,X,X) .  
add(s(X),Y,s(Z)) :- add(X,Y,Z) .  
  
mult(0,_,0) .  
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U) .
```

?- add(X3,Y3,0),mult(X,Y3,s(X3)) .

{X3/0,Y3/0,X4/0}

?- mult(X,0,s(0)) .

{X/s(X5),Y5/0,Z5/s(0)}

?- add(U5,0,s(0)),mult(X5,0,U5) .

{U5/s(X6),Y6/0,Z6/0}

?- add(X6,0,0),mult(X5,0,s(X6)) .

{X6/0,X7/0}

?- mult(X5,0,s(0)) .

Sieht nicht gut aus!

Detailiertere Beschreibung der Erzeugung von Ableitungsbäumen

Eingabe: Anfrage und Programm,
zum Beispiel
`mult(N, M, s(0))` und:

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .  
  
mult(0, _, 0) .  
mult(s(X), Y, Z) :- add(U, Y, Z), mult(X, Y, U) .
```

Ausgabe: Baum, erzeugt durch folgende Schritte:

1. Erzeuge Wurzelknoten mit Anfrage, merke als noch zu bearbeiten.
2. Solange noch zu bearbeitende Knoten vorhanden:
 - wähle linken solchen Knoten
 - ermittle alle Regeln, deren Kopf mit dem linken Literal im Knoten unifizierbar ist
 - erzeuge für jede solche Regel einen (noch weiter zu bearbeitenden) Nachfolgerknoten durch Resolution
 - sortiere Nachfolgerknoten von links nach rechts entsprechend der Reihenfolge verwendeter Regeln von oben nach unten
 - vermerke jeweils verwendeten Unifikator

```
?- mult(N, M, s(0)) .
```

↓ {N/s(X), M/Y, Z/s(0)}

```
?- add(U, Y, s(0)), mult(X, Y, U) .
```

noch zu bearbeiten

Detailiertere Beschreibung der Erzeugung von Ableitungsbäumen

2. Solange noch zu bearbeitende Knoten vorhanden:

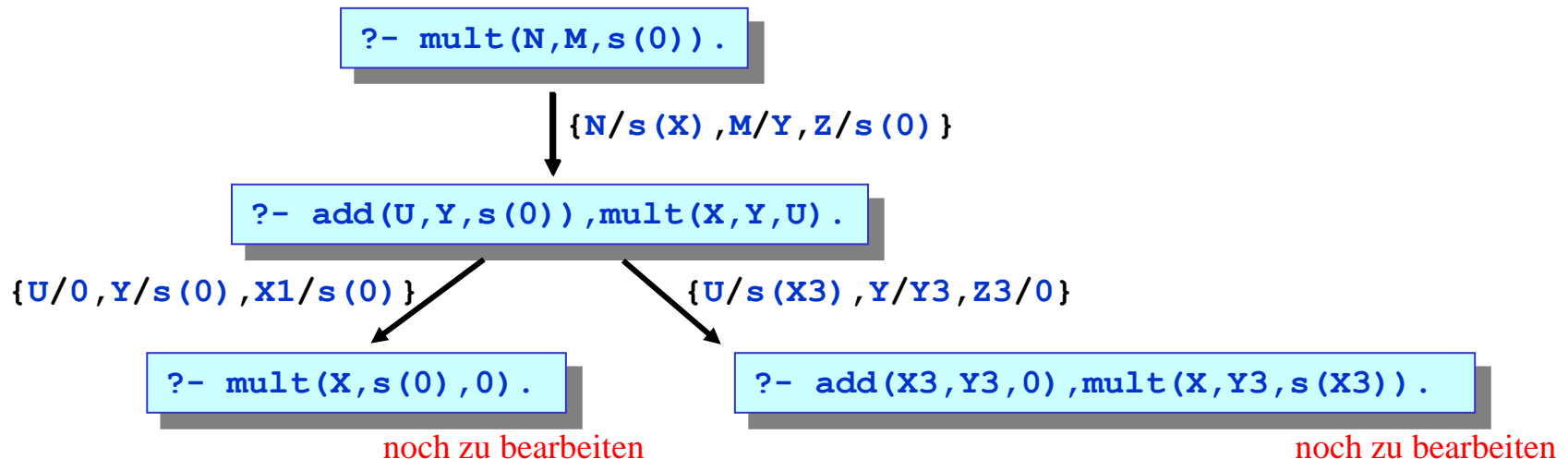
- wähle linken solchen Knoten
- ermittle alle Regeln, deren Kopf mit dem linken Literal im Knoten unifizierbar ist
- erzeuge für jede solche Regel einen (noch weiter zu bearbeitenden) Nachfolgerknoten durch Resolution
- sortiere Nachfolgerknoten von links nach rechts entsprechend der Reihenfolge verwendeter Regeln von oben nach unten
- vermerke jeweils verwendeten Unifikator

```
add(0,X,X).
```

```
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```



```
mult(0,_,0).
```

```
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U).
```



Detailliertere Beschreibung der Erzeugung von Ableitungsbäumen

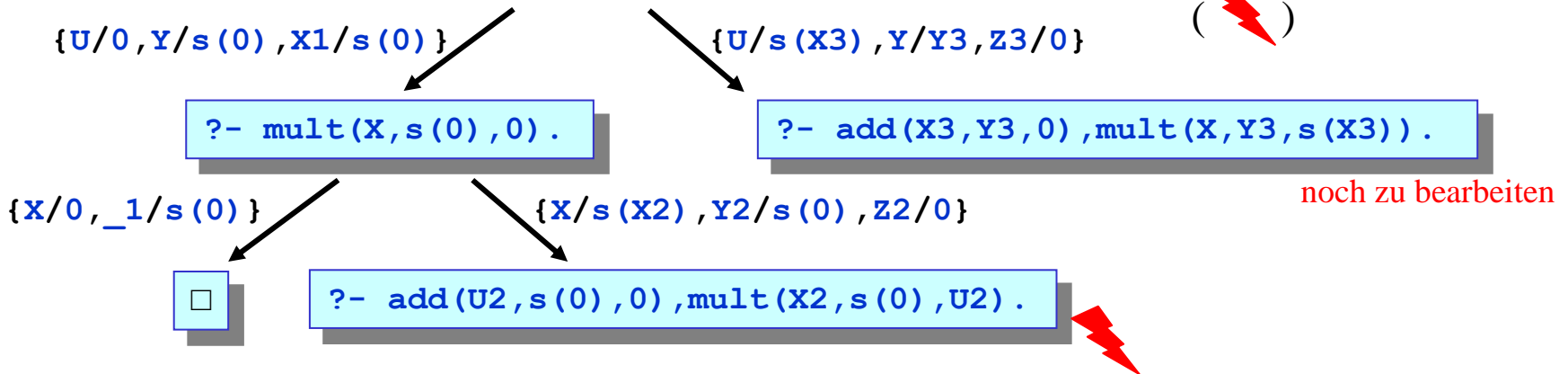
2. Solange noch zu bearbeitende Knoten vorhanden:

- wähle linkensten solchen Knoten
- ermittle alle Regeln, deren Kopf mit dem linkensten Literal im Knoten unifizierbar ist
- erzeuge für jede solche Regel einen (noch weiter zu bearbeitenden) Nachfolgerknoten durch Resolution
- sortiere Nachfolgerknoten von links nach rechts entsprechend der Reihenfolge verwendeter Regeln von oben nach unten
- vermerke jeweils verwendeten Unifikator
- markiere Knoten als nicht weiter zu bearbeiten, wenn leer () oder linkensten Literal mit keinem Regelkopf unifizierbar 

```

add(0, X, X) .
add(s(X), Y, s(Z)) :- add(X, Y, Z) .

mult(0, _, 0) .
mult(s(X), Y, Z) :- add(U, Y, Z), mult(X, Y, U) .
    
```



Detalliertere Beschreibung der Erzeugung von Ableitungsbäumen

2. Solange noch zu bearbeitende Knoten vorhanden:

- wähle linkensten solchen Knoten
- ermittle alle Regeln, deren Kopf mit dem linkensten Literal im Knoten unifizierbar ist
- erzeuge für jede solche Regel einen (noch weiter zu bearbeitenden) Nachfolgerknoten durch Resolution
- sortiere Nachfolgerknoten von links nach rechts entsprechend der Reihenfolge verwendeter Regeln von oben nach unten
- vermerke jeweils verwendeten Unifikator
- markiere Knoten als nicht weiter zu bearbeiten, wenn leer oder linkenstes Literal mit keinem Regelkopf unifizierbar
- an Erfolgsknoten, Annotation der Lösung (Komposition der Unifikatoren, angewandt auf relevante Variablen)

```
add(0, X, X) .
```

```
add(s(X), Y, s(Z)) :- add(X, Y, Z) .
```

```
mult(0, _, 0) .
```

```
mult(s(X), Y, Z) :- add(U, Y, Z), mult(X, Y, U) .
```

```
?- mult(X, s(0), 0) .
```

```
?- add(X3, Y3, 0), mult(X, Y3, s(X3)) .
```

{X/0, _1/s(0)}

{X/s(X2), Y2/s(0), Z2/0}

noch zu bearbeiten

N=s(0),
M=s(0)



```
?- add(U2, s(0), 0), mult(X2, s(0), U2) .
```



Zurück zum Beispiel: Was tun?

```
add(0,X,X) .  
add(s(X),Y,s(Z)) :- add(X,Y,Z) .  
  
mult(0,_,0) .  
mult(s(X),Y,Z) :- add(U,Y,Z),mult(X,Y,U) .
```

?- add(X3,Y3,0),mult(X,Y3,s(X3)) .

{X3/0,Y3/0,X4/0}

?- mult(X,0,s(0)) .

{X/s(X5),Y5/0,Z5/s(0)}

?- add(U5,0,s(0)),mult(X5,0,U5) .

{U5/s(X6),Y6/0,Z6/0}

?- add(X6,0,0),mult(X5,0,s(X6)) .

{X6/0,X7/0}

?- mult(X5,0,s(0)) .

Sieht nicht gut aus!

Versuch: Einfügen eines extra Tests

```
add(0,X,X).  
add(s(X),Y,s(Z)) :- add(X,Y,Z).  
  
mult(0,_,0).  
mult(s(X),Y,Z) :- add(U,Y,Z), Y\=0, mult(X,Y,U).
```

?- mult(N,M,s(0)).

{N/s(X), M/Y, Z/s(0)}

?- add(U,Y,s(0)), Y\=0, mult(X,Y,U).

{U/0, Y/s(0), X1/s(0)}

{U/s(X3), Y/Y3, Z3/0}

?- s(0)\=0, mult(X,s(0),0).

?- add(X3,Y3,0), Y3\=0, mult(X,Y3,s(X3)).

{X/0, _1/s(0)}

{X/s(X2), Y2/s(0), Z2/0}

{X3/0, Y3/0, X4/0}

N=s(0),
M=s(0)

?- add(U2,s(0),0), s(0)\=0,
mult(X2,s(0),U2).

?- 0\=0, mult(X,0,s(0)).



Nur teilweiser Erfolg

```
add(0,X,X).  
add(s(X),Y,s(Z)) :- add(X,Y,Z).  
  
mult(0,_,0).  
mult(s(X),Y,Z) :- add(U,Y,Z), Y\=0, mult(X,Y,U).
```

```
?- mult(N,M,s(s(s(s(0))))).  
N = s(0),  
M = s(s(s(s(0)))) ;  
N = s(s(0)),  
M = s(s(0)) ;  
N = s(s(s(s(0)))) ,  
M = s(0) ;  
false.
```

```
?- mult(s(0),0,0).  
false.
```

Neue Ergebnisse gefunden, alte Ergebnisse verloren!

Erneute „Reparatur“



```
add(0,X,X).  
add(s(X),Y,s(Z)) :- add(X,Y,Z).  
  
mult(0,_,0).  
mult(s(_),0,0).  
mult(s(X),Y,Z) :- add(U,Y,Z),Y\=0,mult(X,Y,U).
```

Jetzt klappt zwar:

```
?- mult(s(0),0,0).  
true.
```

Und es funktioniert
sogar auch allgemein
`mult(?X,?Y,+Z).`

Aber leider (erst hier bemerkt):

```
?- mult(s(0),s(0),N).  
N = s(0) ;  
abort
```

sonst Endlossuche

Also geht nicht mehr
`mult(+X,+Y,?Z).`

Eine neue „Unendlichkeitsfalle“

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(U,Y,Z),Y\=0,mult(X,Y,U).
```

```
?- mult(s(0),s(0),N).
```

{X/0,Y/s(0),N/Z}

```
?- add(U,s(0),Z),s(0)\=0,mult(0,s(0),U).
```

{U/0,X1/s(0),Z/s(0)}

{U/s(X2),Y2/s(0),Z/s(Z2)}

```
?- s(0)\=0,mult(0,s(0),0).
```

```
?- add(X2,s(0),Z2),s(0)\=0,mult(0,s(0),s(X2)).
```

{_1/s(0)}

N=s(0)

wichtige Beobachtung:
(siehe letzte Vorlesung)

```
?- add(U,s(0),Z).
U = 0, Z = s(0) ;
U = s(0), Z = s(s(0)) ;
...
```

vs.

```
?- add(s(0),U,Z).
Z = s(U).
```

Sieht nicht gut aus!

Ausnutzen von Kommutativität

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .  
  
mult(0, _, 0) .  
mult(s(_), 0, 0) .  
mult(s(X), Y, Z) :- add(Y, U, Z), Y \= 0, mult(X, Y, U) .
```

wichtige Beobachtung:
(siehe letzte Vorlesung)

```
?- add(U, s(0), Z) .  
U = 0, Z = s(0) ;  
U = s(0), Z = s(s(0)) ;  
...
```

vs.

```
?- add(s(0), U, Z) .  
Z = s(U) .
```

Ausnutzen von Kommutativität

```
add(0, X, X) .  
add(s(X), Y, s(Z)) :- add(X, Y, Z) .  
  
mult(0, _, 0) .  
mult(s(_), 0, 0) .  
mult(s(X), Y, Z) :- add(Y, U, Z), Y \= 0, mult(X, Y, U) .
```

```
?- mult(s(0), s(0), N) .
```

{X/0, Y/s(0), N/Z} ↓

```
?- add(s(0), U, Z), s(0) \= 0, mult(0, s(0), U) .
```

{X1/0, U/Y1, Z/s(Z1)} ↓

```
?- add(0, Y1, Z1), s(0) \= 0, mult(0, s(0), Y1) .
```

{Y1/X2, Z1/X2} ↓

```
?- s(0) \= 0, mult(0, s(0), X2) .
```

{_1/s(0), X2/0} ↓

□ N=s(0)

Eine tatsächlich allgemein geeignete Definition

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),Y,Z) :- add(Y,U,Z),Y\=0,mult(X,Y,U).
```

```
?- mult(N,M,s(s(s(s(0))))).
N = s(0),
M = s(s(s(s(0)))) ;
N = s(s(0)),
M = s(s(0)) ;
N = s(s(s(s(0)))) ,
M = s(0) ;
false.

?- mult(s(0),s(0),N).
N = s(0).

?- add(X,0,X),not(mult(s(s(_)),s(s(_)),X)).
...
```

Es funktionieren alle
Aufrufmodi außer
`mult(?X,?Y,?Z)!`

Die operationelle Semantik:

- bildet den tatsächlichen Prolog-Suchvorgang ab, mit Backtracking
- benutzt essentiell Unifikation (und Resolution)
- erlaubt Verstehen von Effekten wie Nichttermination
- gibt Einblick in Auswirkungen von Änderungen der Reihenfolge von und innerhalb Regeln

Deskriptive Programmierung

Negation in Prolog

Negation (1)

- Der logischen Programmierung liegt zunächst eine positive Logik zugrunde.

Ein Literal ist beweisbar, wenn es (ggfs. über mehrere Schritte) auf die Beweisbarkeit unmittelbarer Tatsachen zurückgeführt werden kann.

- Prolog bietet aber auch die Möglichkeit, **Negation** zu verwenden.
 - Diese ist allerdings nur bedingt mit der erwartbaren logischen Bedeutung vereinbar.
 - `\+ Goal`, bzw. `not (Goal)`, ist beweisbar gdw. `Goal` nicht beweisbar ist.

Beispiel: `\+ member (4, [2, 3])` ist beweisbar, da `member (4, [2, 3])` nicht beweisbar ist, d.h. es existiert ein „endlicher Misserfolgsbaum“.

Vorsicht:

```
?- member (X, [2, 3]) .           => X = 2; X = 3.
?- \+ member (X, [2, 3]) .       => false.
?- \+ \+ member (X, [2, 3]) .    => true.
```

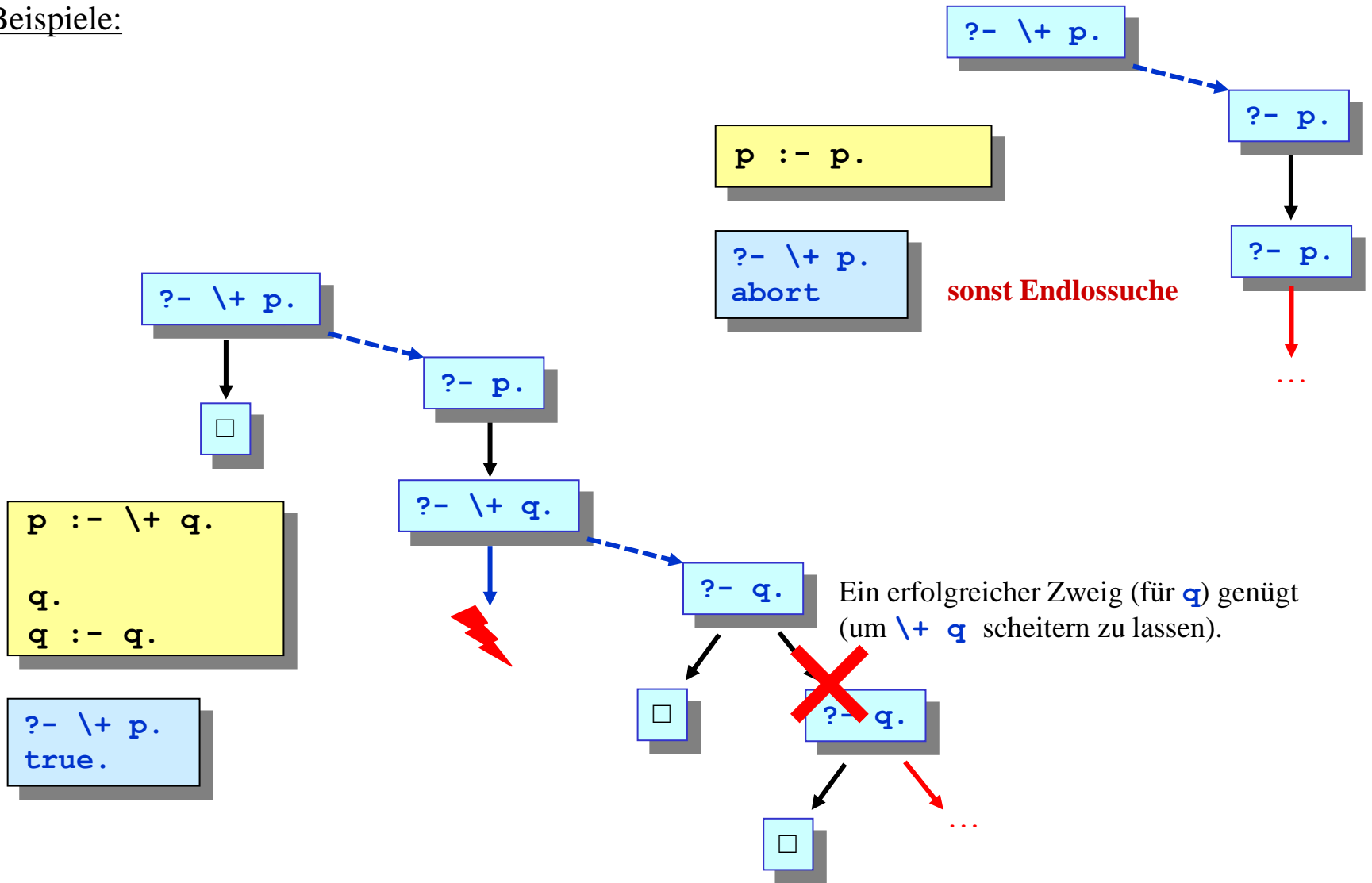
(Negation führt nicht zur Bindung von Variablen.)

- Warum „endlicher Misserfolgsbaum“?
 - Wir können nicht allgemein zeigen, dass aus den Regeln eines Programms eine bestimmte negative Aussage folgt.
 - Wir können lediglich zeigen, dass wir eine bestimmte positive Aussage nicht folgern können. (Negation as Failure)
 - Dabei bedeutet „zeigen“, einen Beweis zu suchen und zu scheitern.
 - Dass wir wirklich notgedrungen scheitern, lässt sich nur mit Sicherheit sagen, wenn der Suchraum endlich ist.
- Zu Grunde liegende Annahme:

Closed World Assumption

Negation (3)

Beispiele:



Negation (4)

Beispiele mit Variablen:

```
human(marcellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X) :- human(X), \+ married(X,Y).
```

```
?- single(X).
X = marcellus.
```

```
?- single(marcellus).
true.
```

```
?- single(vincent).
false.
```

```
human(marcellus).
human(vincent).
human(mia).

married(vincent,mia).
married(mia,vincent).

single(X) :- \+ married(X,Y), human(X).
```

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

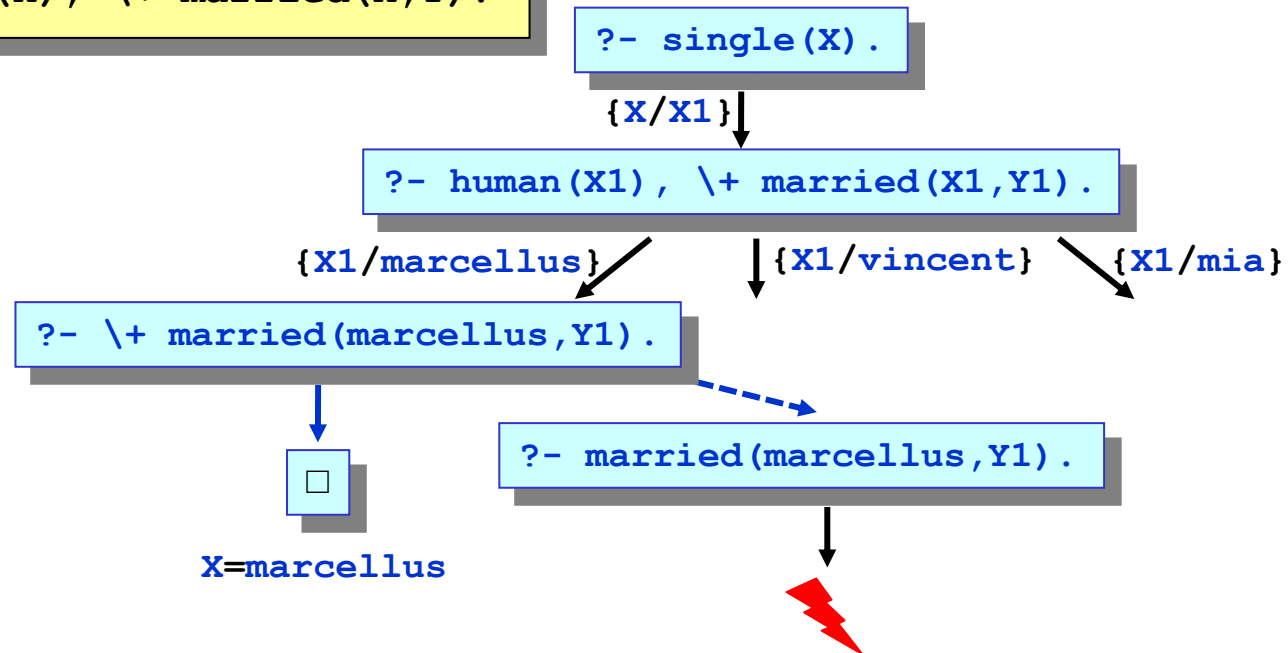
```
?- single(marcellus).
true.
```

```
?- single(vincent).
false.
```

Negation (5)

Beispiele mit Variablen:

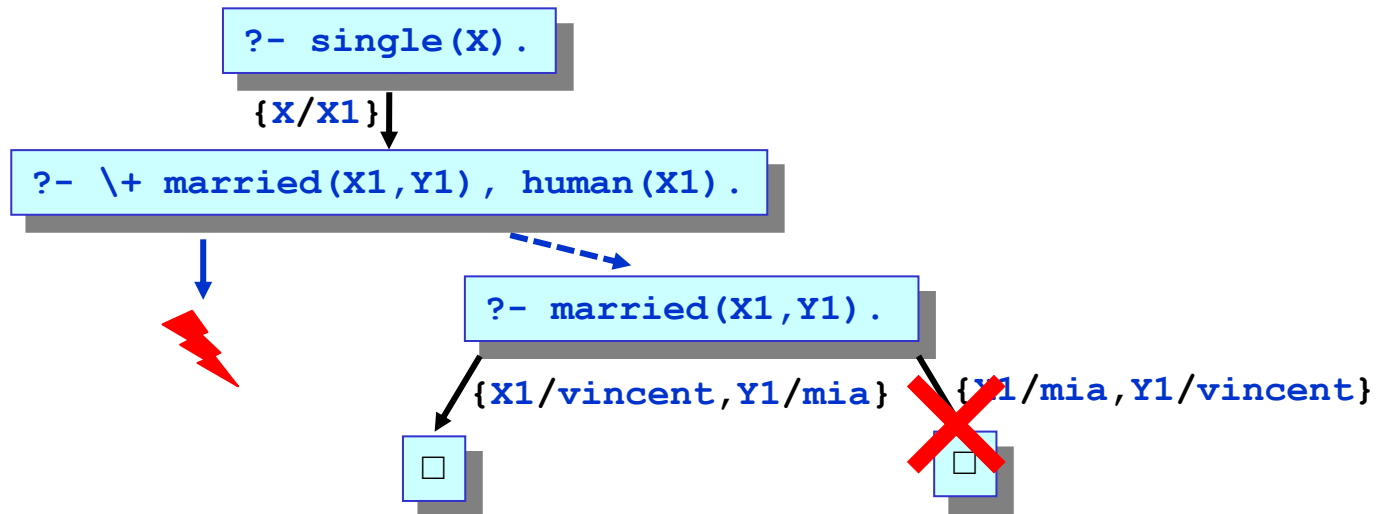
```
human(marcellus).  
human(vincent).  
human(mia).  
  
married(vincent,mia).  
married(mia,vincent).  
  
single(X) :- human(X), \+ married(X,Y).
```



Negation (6)

Beispiele mit Variablen:

```
human(marcellus) .  
human(vincent) .  
human(mia) .  
  
married(vincent,mia) .  
married(mia,vincent) .  
  
single(X) :- \+ married(X,Y) , human(X) .
```



Negation (7)

Beispiele mit Variablen:

```
human(marcellus) .  
human(vincent) .  
human(mia) .  
  
married(vincent,mia) .  
married(mia,vincent) .  
  
single(X) :- \+ married(X,Y), human(X) .
```

```
?- single(marcellus) .
```

{X1/marcellus} ↓

```
?- \+ married(marcellus,Y1), human(marcellus) .
```

```
?- human(marcellus) .
```



```
?- married(marcellus,Y1) .
```



Negation (8)

Erklärung aus „logischer Sicht“:

Unter den Annahmen, dass **x** ursprünglich ungebunden ist, und durch **human (x)** stets gebunden wird, bedeutet

```
single(x) :- human(x), \+ married(x,y).
```

dass $\forall X : \text{human}(X) \wedge \neg(\exists Y : \text{married}(X,Y)) \Rightarrow \text{single}(X)$.

Unter den gleichen Annahmen bedeutet jedoch


```
single(x) :- \+ married(x,y), human(x).
```

dass $\forall X : \neg(\exists X,Y : \text{married}(X,Y)) \wedge \text{human}(X) \Rightarrow \text{single}(X)$.

- keine echte logische Negation: stattdessen Negation as Failure
- Beweissuche in „Nebenrechnung“, bindet nach außen keine Variablen
- lässt sich nur prozedural/operationell (wirklich) verstehen
- Probleme bei Versuch deklarativer Lesart:
 - nicht kompositionell/substitutiv
 - sensitiv gegenüber Änderungen der Reihenfolge innerhalb Regeln
 - T_P -Operator wäre nicht-monoton

Deskriptive Programmierung

Der Cut-Operator

- Die operationelle **Backtracking-Methode** von Prolog:
 - merkt sich jeden Punkt, an dem noch weitere Alternativen (durch andere Regeln) besucht werden könnten.
 - kann dadurch einen hohen Verwaltungs- und Ausführungsaufwand erfordern.
- Durch Verwendung des **„Cut“-Operators**: 
 - kann das Backtracking explizit beeinflusst werden,
 - nämlich das Ausprobieren bestimmter **Alternativen verhindert** werden, indem Teile des Ableitungsbaums „abgeschnitten“ werden.
 - können daher die Laufzeit und der Speicherbedarf verringert werden.
 - können Programme unter Umständen „einfacher“/kürzer werden.
 - können **korrekte** Antworten unter Umständen nicht mehr gefunden werden.

Der „Cut“-Operator (2)

- Der Cut wird in Anfragen oder Regelrümpfen wie ein Literal mit dem 0-stelligen Prädikat **!** notiert.
- Die Bedeutung des „Cut“-Operators kann nur **operationell** angegeben werden.

- Beispiel:

$$\begin{array}{l} p(x) \text{ :- } q(x), !, s(x) . \\ p(x) \text{ :- } r(x) . \end{array}$$

- Kann $q(x)$ **nicht** bewiesen werden, wird für $p(x)$ die nächste Klausel ausprobiert.
- Wird $q(x)$ **einmal** bewiesen, wird wegen **!** im Fall eines späteren Fehlschlags, in $s(x)$, kein weiterer Beweis für $q(x)$ gesucht. Es wird dann auch keine weitere Regel für $p(x)$ mehr ausprobiert.

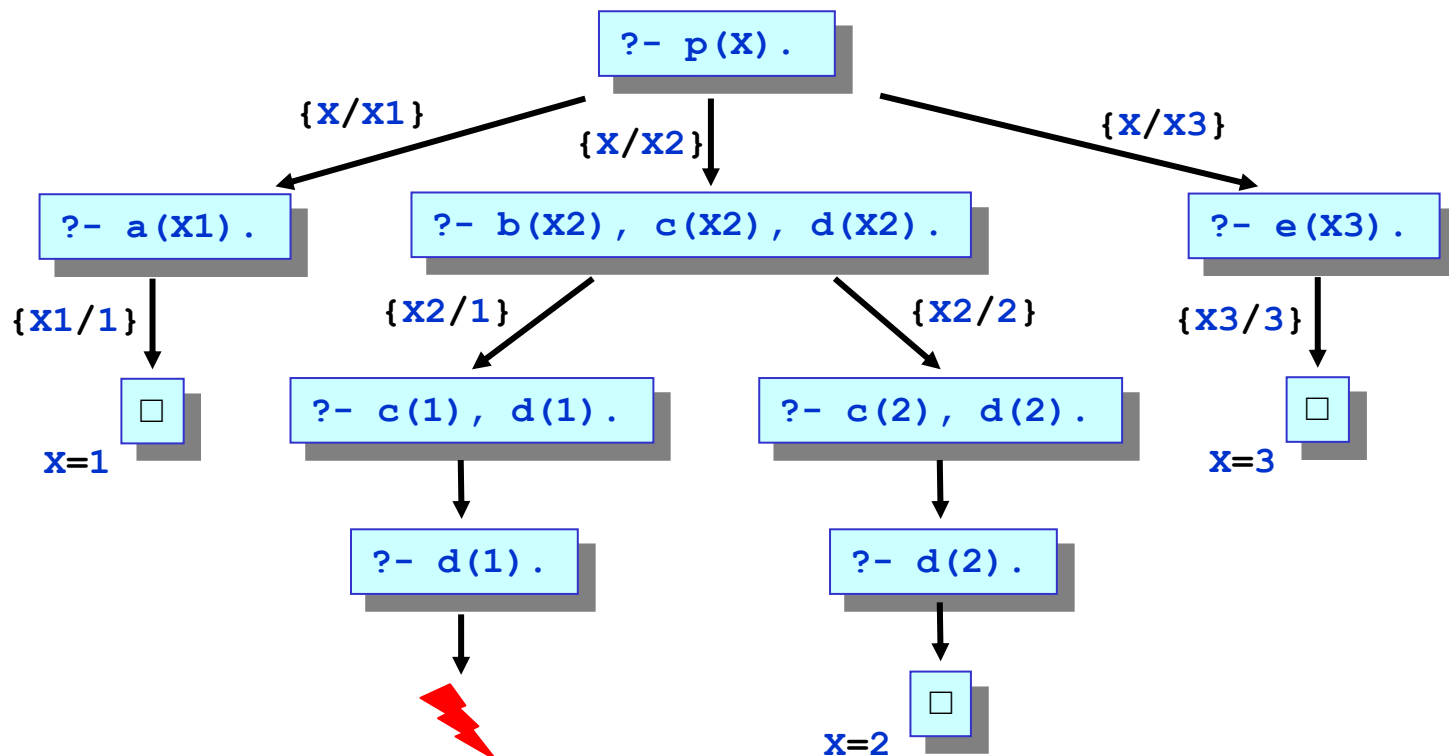
Also, ist $s(x)$ mit der durch $q(x)$ ermittelten Belegung für x nicht beweisbar, dann ist auch $p(x)$ nicht beweisbar.

- Allgemein:
 - Der Cut selbst als Teilziel ist immer erfolgreich/erfüllt.
 - Wenn ein anderes Teilziel eine Regel mit Cut benutzt, und dieser Cut im Verlauf der Ableitung erreicht wird, dann sind alle gemachten Entscheidungen seit (und inklusive) Wahl der entsprechenden Regel unumkehrbar.
 - Entscheidungen die nach Antreffen des Cut anstehen, werden jedoch mit all ihren Alternativen untersucht.
 - Und wenn es zum ursprünglichen Teilziel noch Alternativen gab, dann werden diese ebenfalls weiter untersucht.

Der „Cut“-Operator (4)

Illustration an Hand von Ableitungsbäumen:

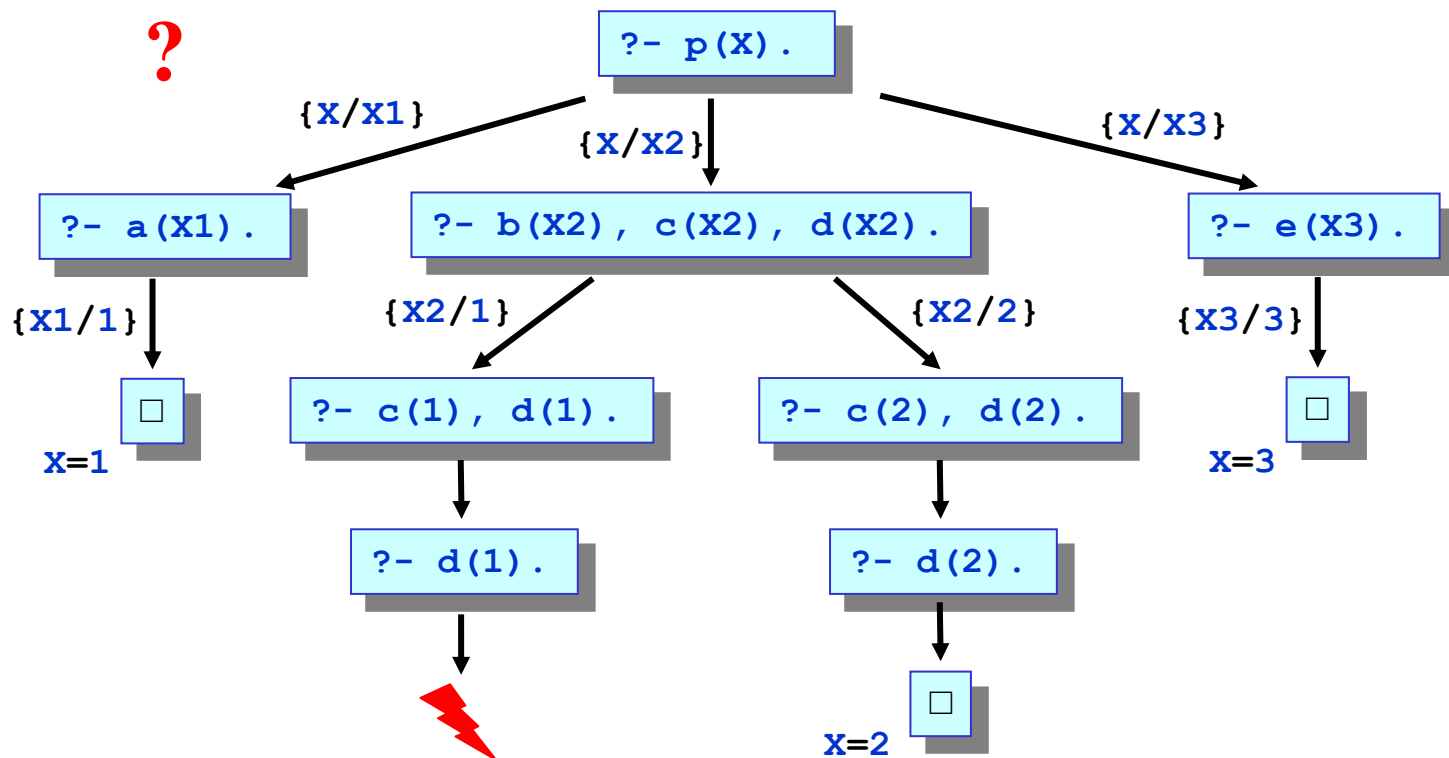
```
p(X) :- a(X) .  
p(X) :- b(X), c(X), d(X) .  
p(X) :- e(X) .  
  
a(1) . b(1) . b(2) . c(1) . c(2) . d(2) . e(3) .
```



Der „Cut“-Operator (4)

Illustration an Hand von Ableitungsbäumen:

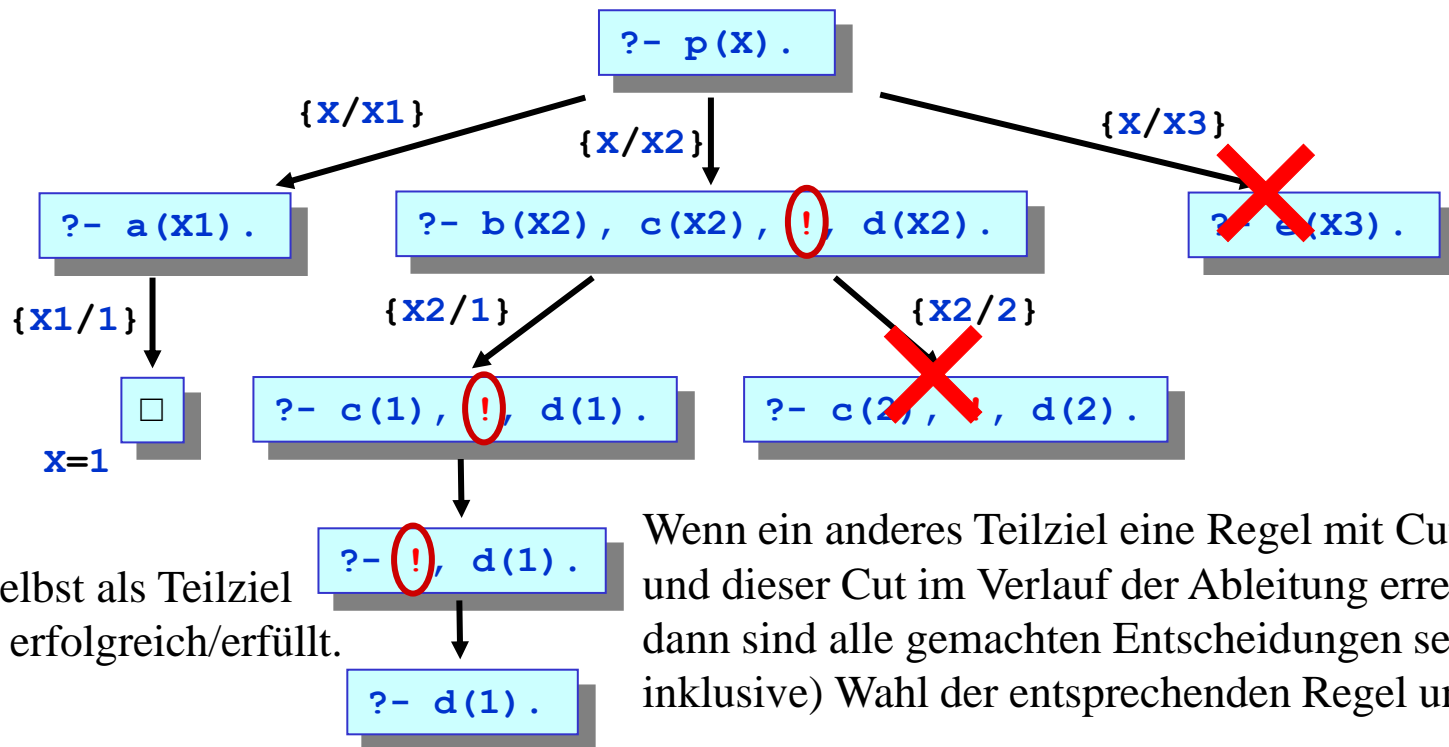
```
p(X) :- a(X) .  
p(X) :- b(X), c(X), !, d(X) .  
p(X) :- e(X) .  
  
a(1) . b(1) . b(2) . c(1) . c(2) . d(2) . e(3) .
```



Der „Cut“-Operator (4)

Illustration an Hand von Ableitungsbäumen:

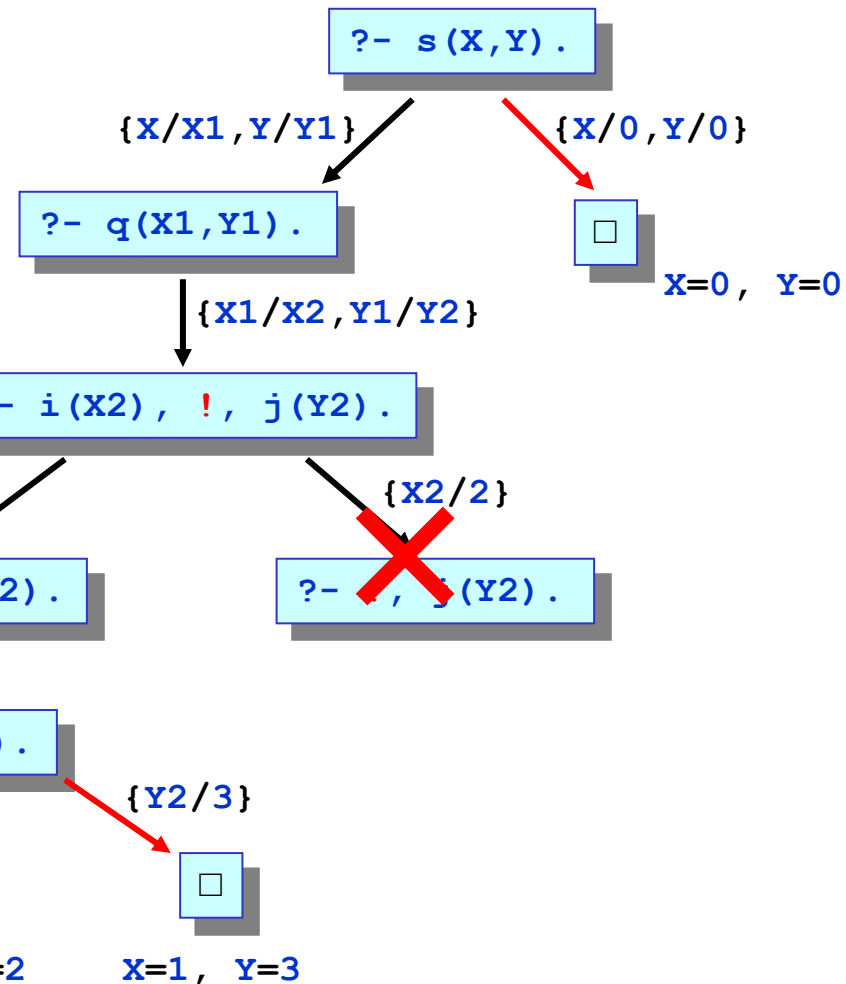
```
p(X) :- a(X) .  
p(X) :- b(X), c(X), !, d(X) .  
p(X) :- e(X) .  
  
a(1) . b(1) . b(2) . c(1) . c(2) . d(2) . e(3) .
```



Der „Cut“-Operator (5)

Illustration an Hand von Ableitungsbäumen:

```
s(X,Y) :- q(X,Y).  
s(0,0).  
  
q(X,Y) :- i(X), !, j(Y).  
i(1). i(2). j(1). j(2). j(3).
```



Entscheidungen die nach Antreffen des Cut anstehen, werden mit all ihren Alternativen untersucht.

Und wenn es zum ursprünglichen Teilziel noch Alternativen gab, dann werden diese ebenfalls weiter untersucht.

Der „Cut“-Operator (6)

```
s(X,Y) :- q(X,Y).  
s(0,0).  
  
q(X,Y) :- i(X), !, j(Y).  
  
i(1). i(2). j(1). j(2). j(3).
```

```
?- s(X,Y).  
X = 1, Y = 1;  
X = 1, Y = 2;  
X = 1, Y = 3;  
X = 0, Y = 0.
```

vs.

```
?- s(2,1).  
true.  
  
?- s(2,2).  
true.  
  
?- s(2,3).  
true.
```

Deklarative Semantik ohne Cut:

$$T_P(\emptyset) = \{s(0,0), i(1), i(2), j(1), j(2), j(3)\}$$

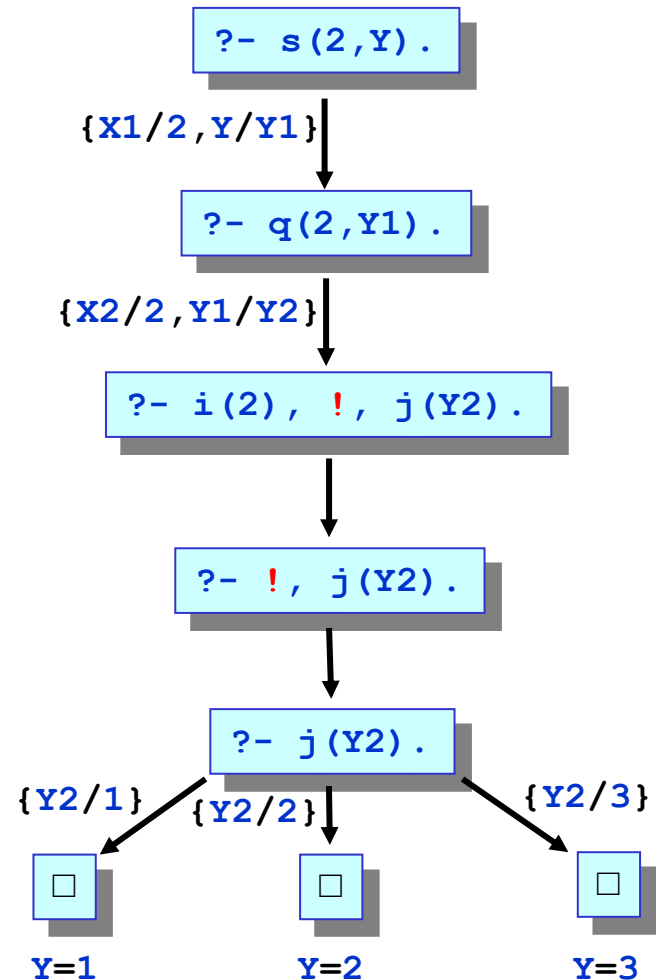
$$T_P(T_P(\emptyset)) = T_P(\emptyset) \cup \{q(1,1), q(1,2), q(1,3), \\ q(2,1), q(2,2), q(2,3)\}$$

$$T_P(T_P(T_P(\emptyset))) = T_P(T_P(\emptyset)) \cup \{s(1,1), s(1,2), s(1,3), \\ s(2,1), s(2,2), s(2,3)\}$$

Der „Cut“-Operator (7)

```
s(X,Y) :- q(X,Y).  
s(0,0).  
  
q(X,Y) :- i(X), !, j(Y).  
  
i(1). i(2). j(1). j(2). j(3).
```

```
?- s(2,1).  
true.  
  
?- s(2,2).  
true.  
  
?- s(2,3).  
true.
```



Der „Cut“-Operator – Sinnvolle Verwendung (1)

- Der Cut-Operator kann verwendet werden, um „unnötige“ Vergleiche abzukürzen.
- Zum Beispiel, in Haskell:

```
nodups [] = []  
nodups (x : xs) | elem x xs = nodups xs  
                | otherwise = x : nodups xs
```



```
nodups ([], []) .  
nodups ([X|Xs], Ys) :- member (X, Xs), nodups (Xs, Ys) .  
nodups ([X|Xs], [X|Ys]) :- not (member (X, Xs)), nodups (Xs, Ys) .
```

Effizienter:

```
nodups ([], []) .  
nodups ([X|Xs], Ys) :- member (X, Xs), !, nodups (Xs, Ys) .  
nodups ([X|Xs], [X|Ys]) :- nodups (Xs, Ys) .
```

Der „Cut“-Operator – Sinnvolle Verwendung (2)

- Der Cut-Operator kann verwendet werden, um „unnötige“ Vergleiche abzukürzen:

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



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



?



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

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

- Für bestimmte Aufrufe verhält sich die linke Variante sehr unglücklich. (Für welche?)

Der „Cut“-Operator – Bedingte Anweisung

- Allgemein kann der Cut-Operator verwendet werden, um „bedingte Anweisungen“ zu realisieren:

```
p :- q, !, s.  
p :- r.
```

kann interpretiert werden als:

```
if q then s else r
```

- Prolog bietet für diese Art Fallunterscheidung eine abkürzende Notation an:

```
p :- q -> s ; r.
```

Jeder der Zweige kann Backtracking durchführen, jedoch nicht der Test selbst (nach erstem Erfülltsein)!

Beispiel:

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



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

Der „Cut“-Operator kann zur Implementierung der Negation verwendet werden:

```
not(X) :- call(X), !, fail.  
not(X).
```

- **call(X)** ist beweisbar, wenn **X** mit einem Term instantiiert ist, der in einer Anfrage vorkommen darf, und die Anfrage **?- X.** beweisbar ist.
ein Metaprädikat (nimmt anderes Prädikat/Literal/Anfrage als Argument)
- **fail** ist ein vordefiniertes Prädikat, für das keine Klauseln existieren, und das daher nie beweisbar ist.

Vorteile Cut:

- kann Effizienz von Programmen erhöhen
- kann Formulierung von Prädikaten vereinfachen

Nachteile Cut:

- lässt sich nur operationell, nicht deklarativ, verstehen
- eine Hauptquelle von Fehlern in Prolog-Programmen (wegen fehlender und/oder falscher Antworten)

Deskriptive Programmierung

Ein- und Ausgabe in Prolog

Ein kleines „interaktives“ Programm (1)

Zugleich mal ein Beispiel für ein etwas „praktischeres“ Prolog-Programm ...

Ein kleines numerisches Spiel:

- es gibt zwei Spieler: **A** und **B**
- zu Beginn liegen Karten mit den Zahlen **1** bis **9** offen bereit
- die Spieler wählen abwechselnd eine Karte
- gewonnen hat, wer zuerst drei Karten besitzt, deren Summe **15** ergibt

Beispiel ... („live“)

Ein kleines „interaktives“ Programm (2)

Ein Hilfsprädikat:

```
move(Xs,X,Ys) :- member(X,Xs), delete(Xs,X,Ys).
```

```
?- move([1,2,3,4,5,6,7,8,9],5,Ys).  
Ys = [1,2,3,4,6,7,8,9] ;  
false.
```

„Hauptschleife“:

```
play(As,Bs,Xs) :- read(A), !, move(Xs,A,Ys), A1=[A|As],  
                  print((Bs,A1,Ys)), nl,  
                  (sum15(A1); Ys=[]; play(Bs,A1,Ys)).
```

Benutzt (genaue Definition nicht so wesentlich, hier jetzt nicht besonders elegant):

```
sum15(As) :- move(As,A1,Xs), move(Xs,A2,Ys), move(Ys,A3,_),  
            A is A1+A2+A3, A=15.
```

Ein kleines „interaktives“ Programm (3)

Etwas besser „verpackt“:

```
play(As,Bs,Xs) :- read(A), !, move(Xs,A,Ys), A1=[A|As],
                  show(Bs,A1,Ys),
                  (sum15(A1); Ys=[]; play(Bs,A1,Ys)).
```

```
start :- As=[], Bs=[], Xs=[1,2,3,4,5,6,7,8,9],
         show(As,Bs,Xs), play(As,Bs,Xs).
```

```
show(As,Bs,Xs) :- print((As,Bs,Xs)), nl.
```

Nun würden wir gern gegen einen „intelligenten“ Gegner spielen!

Idee: ein Prädikat `to_win(As,Bs,Xs,X)` schreiben, das nächsten Zug ermittelt.

Versuch intelligenten Spielens

Ein optimaler Zug für **A** siegt entweder sofort oder erzwingt Niederlage von **B**:

```
to_win(As,Bs,Xs,X) :- move(Xs,X,Ys), A1 = [X|As],
                    (sum15(A1); will_lose(Bs,A1,Ys)).
```

Eine Situation ist aussichtslos für **B**, wenn kein Unentschieden erreicht, und kein Zug zum direkten Sieg von **B** führt oder zumindest zu einer Situation, in der **A** nicht gewinnen kann:

```
will_lose(Bs,As,Ys) :- Ys\=[], not((move(Ys,Y,Zs), B1=[Y|Bs],
                                   (sum15(B1); not(to_win(As,B1,Zs,_))
                                   ))).
```

```
?- to_win([3,9],[4,8],[1,2,5,6,7],X).
X = 5 ;
false.

?- to_win([],[],[1,2,3,4,5,6,7,8,9],A).
false.

?- move([1,2,3,4,5,6,7,8,9],A,Xs),to_win([], [A],Xs,B).
false.
```


Intelligentes Spielen, Inkaufnahme von Unentschieden

Um auszudrücken, dass für **A** bestenfalls, und tatsächlich, ein Unentschieden erzwingbar ist:

```
to_draw(As,Bs,Xs,X) :- not(to_win(As,Bs,Xs,_)),
                        move(Xs,X,Ys), A1=[X|As],
                        (Ys=[]; to_draw(Bs,A1,Ys,_)).
```

```
?- to_draw([],[],[1,2,3,4,5,6,7,8,9],A).
A = 1 ;
...

?- move([1,2,3,4,5,6,7,8,9],A,Xs),not(to_draw([], [A],Xs,B)).
false.

?- to_draw([4,8],[3,5,9],[1,2,6,7],X).
false.

?- to_draw([8,9],[1,4,7],[2,3,5,6],X).
X = 5 ;
X = 6 ;
X = 6 ;
X = 6 ;
X = 6 ;
false.
```

Unterscheidung zwischen Spieler und Gegenspieler:

```
play(As,Bs,Xs) :- read(A), !, move(Xs,A,Ys), A1=[A|As],
                  show(Bs,A1,Ys),
                  (sum15(A1); Ys=[]; play(Bs,A1,Ys)).
```



```
play(As,Bs,Xs) :- read(A), !, move(Xs,A,Ys), A1=[A|As],
                  %show(Bs,A1,Ys),
                  (sum15(A1); Ys=[]; reply(Bs,A1,Ys)).

reply(Bs,As,Ys) :- (to_win(Bs,As,Ys,B); to_draw(Bs,As,Ys,B)),
                  !, move(Ys,B,Zs), B1=[B|Bs],
                  show(As,B1,Zs),
                  (sum15(B1); Zs=[]; play(As,B1,Zs)).
```

Etwas Optimierung:

```
to_draw(As,Bs,Xs,X) :- not(to_win(As,Bs,Xs,_)),  
                        move(Xs,X,Ys), A1=[X|As],  
                        (Ys=[]; to_draw(Bs,A1,Ys,_)).
```



```
to_draw(As,Bs,Xs,X) :- move(Xs,X,Ys), A1=[X|As],  
                        (Ys=[]; not(to_win(Bs,A1,Ys,_))),  
                        to_draw(Bs,A1,Ys,_).
```



```
to_draw(As,Bs,Xs,X) :- move(Xs,X,Ys), A1=[X|As],  
                        (Ys=[]; not(to_win(Bs,A1,Ys,_))).
```

Live-Demonstration ...

„Hausaufgabe“: Implementieren Sie Tic-Tac-Toe (mit „intelligentem“ Gegenspieler). (?)

Deskriptive Programmierung

Prolog-Spracherweiterung: DCGs

- Angenommen, wir wollen Sätze der englischen Sprache modellieren.
- Wir brauchen verschiedene Kategorien von Worten und Satzteilen:

verb, noun, verb phrase, ...

sowie Regeln zur grammatikalisch richtigen Kombination derselben:

sentence	→	noun phrase, verb phrase
noun phrase	→	determiner, noun
verb phrase	→	verb, noun phrase
	...	

- Und natürlich einen Mechanismus, eine solche Grammatik „auszuwerten“.

Symbolische Sprachverarbeitung/-repräsentation (2)

Einfache Umsetzung in Prolog:

- Wortkategorien + Regeln:

```
det([the]).  
det([a]).  
  
n([woman]).  
n([man]).  
  
v([knows]).
```

```
np(Z) :- det(X), n(Y), append(X,Y,Z).  
  
vp(Z) :- v(X), np(Y), append(X,Y,Z).  
vp(Z) :- v(Z).  
  
s(Z) :- np(X), vp(Y), append(X,Y,Z).
```

- Verwendung:

```
?- s([a,woman,knows,a,man]).  
true.  
  
?- s([the,woman,knows]).  
true.  
  
?- s(Z).  
Z = [the, woman, knows, the, woman] ;  
...  
Z = [a, man, knows].
```

Schön, aber
potentiell
ineffizient
wegen der
Art der
Verwendung
von **append**!

Symbolische Sprachverarbeitung/-repräsentation (3)

Verwendung von Akkumulatoren/„Differenzlisten“:

```
det([the]).  
det([a]).  
  
n([woman]).  
n([man]).  
  
v([knows]).
```



```
det([the|U],U).  
det([a|U],U).  
  
n([woman|U],U).  
n([man|U],U).  
  
v([knows|U],U).
```

```
np(Z) :- det(X), n(Y), append(X,Y,Z).  
  
vp(Z) :- v(X), np(Y), append(X,Y,Z).  
vp(Z) :- v(Z).  
  
s(Z) :- np(X), vp(Y), append(X,Y,Z).
```



```
np(ZU,U) :- det(ZU,YU), n(YU,U).  
  
vp(ZU,U) :- v(ZU,YU), np(YU,U).  
vp(ZU,U) :- v(ZU,U).  
  
s(ZU,U) :- np(ZU,YU), vp(YU,U).
```

Symbolische Sprachverarbeitung/-repräsentation (4)

Neue Version:

```
det([the|U],U).  
det([a|U],U).  
  
n([woman|U],U).  
n([man|U],U).  
  
v([knows|U],U).
```

```
np(ZU,U) :- det(ZU,YU), n(YU,U).  
  
vp(ZU,U) :- v(ZU,YU), np(YU,U).  
vp(ZU,U) :- v(ZU,U).  
  
s(ZU,U) :- np(ZU,YU), vp(YU,U).
```

Tests:

```
?- s([a,woman, knows ,a,man], []).  
true.  
  
?- s([the,woman, knows], []).  
true.  
  
?- s(Z, []).  
Z = [the, woman, knows, the, woman] ;  
...  
Z = [a, man, knows].
```


Spezielles Prolog-Feature: „Definite Clause Grammars“

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

```
n --> [woman].  
n --> [man].
```

```
v --> [knows].
```

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

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

```
vp --> v.
```

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

Automatische Umsetzung:

```
?- listing.  
v([knows|A], A).  
np(A, C) :- det(A, B), n(B, C).  
det([the|A], A).  
det([a|A], A).  
n([woman|A], A).  
n([man|A], A).  
s(A, C) :- np(A, B), vp(B, C).  
vp(A, C) :- v(A, B), np(B, C).  
vp(A, B) :- v(A, B).
```

Bisher können wir nur testen oder generieren:

```
?- s([a,woman, knows ,a ,man] , []).  
true.  
  
?- s(Z, []).  
Z = [the, woman, knows, the, woman] ;  
...  
Z = [a, man, knows].
```

Zusätzlich würden wir gerne echt „parsen“, also mit Ausgabe der Satzstruktur.

Durch Hinzufügen eines Syntaxbaum-Arguments:

```
det(td) --> [the].  
det(td) --> [a].  
  
n(tn) --> [woman].  
n(tn) --> [man].  
  
v(tv) --> [knows].
```

```
np(tnp(T,S)) --> det(T), n(S).  
  
vp(tv(T,S)) --> v(T), np(S).  
vp(tv(T)) --> v(T).  
  
s(ts(T,S)) --> np(T), vp(S).
```

Symbolische Sprachverarbeitung/-repräsentation (7)

```
det(td) --> [the].  
det(td) --> [a].
```

```
n(tn) --> [woman].  
n(tn) --> [man].
```

```
v(tv) --> [knows].
```

```
np(tnp(T,S)) --> det(T), n(S).
```

```
vp(tvp(T,S)) --> v(T), np(S).
```

```
vp(tvp(T)) --> v(T).
```

```
s(ts(T,S)) --> np(T), vp(S).
```

```
?- s(T, [a, woman, knows, a, man], []).  
T = ts(tnp(td, tn), tvp(tv, tnp(td, tn))).
```

```
?- s(T, Z, []).  
T = ts(tnp(td, tn), tvp(tv, tnp(td, tn))),  
Z = [the, woman, knows, the, woman] ;  
...  
T = ts(tnp(td, tn), tvp(tv)),  
Z = [a, man, knows].
```

```
?- listing(s).  
s(ts(A, C), B, E) :- np(A, B, D), vp(C, D, E).
```

Eine weitere sinnvolle Verwendung von zusätzlichen Argumenten:
grammatikalische Features.

- Angenommen, wir wollen Pronomen einführen:

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

```
n --> [woman].  
n --> [man].
```

```
v --> [knows].
```

```
pro --> [he].  
pro --> [she].  
pro --> [him].  
pro --> [her].
```

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

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

```
vp --> v.
```

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

- **Hmm:**

```
?- s(Z, []).  
Z = [he, knows, he] ;  
Z = [he, knows, she] ; ...
```

- Korrektur mittels zusätzlicher Argumente:

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

n --> [woman].
n --> [man].

v --> [knows].

pro(subject) --> [he].
pro(subject) --> [she].
pro(object) --> [him].
pro(object) --> [her].
```

```
np(X) --> pro(X).
np(  ) --> det, n.

vp --> v, np(object).
vp --> v.

s --> np(subject), vp.
```

- Nun:

```
?- s(Z, []).
Z = [he, knows, him] ;
Z = [he, knows, her] ;
Z = [he, knows, the, woman] ;
Z = [he, knows, the, man] ;
Z = [he, knows, a, woman] ; ...
```

Fallstudie: Parsen arithmetischer Ausdrücke

- Zur Erinnerung:

```
expr ::= term + expr | term
term ::= factor * term | factor
factor ::= nat | (expr)
```

- Umsetzung in Haskell:

```
expr :: Parser Expr
expr = (Add <$> term <* char '+' <*> expr) ||| term

term :: Parser Expr
term = (Mul <$> factor <* char '*' <*> term) ||| factor

factor :: Parser Expr
factor = (Lit <$> nat) ||| (char '(' *> expr <* char ')')
```

Fallstudie: Parsen arithmetischer Ausdrücke

- Nun in Prolog:

```
expr(+ (T,E) ) --> term(T) , "+" , expr(E) .
expr(T)         --> term(T) .

term(* (F,T) ) --> factor(F) , "*" , term(T) .
term(F)         --> factor(F) .

factor(N) --> nat(N) .
factor(E) --> "(" , expr(E) , ")" .

nat(0) --> "0" .
...
nat(9) --> "9" .
```

(So, mit expliziten Strings, werden wir DCGs auch in der Übung machen.)

- Tests:

```
?- expr(E,"1+2*3",""), R is E.
E = 1+2*3, R = 7.

?- expr((1+2)*3,S,"").
S = [40, 49, 43, 50, 41, 42, 51] ;

?- expr((1+2)*3,S,""), writef("%s",[S]).
(1+2)*3
```

Fallstudie: Parsen arithmetischer Ausdrücke

- Ausnutzung verschiedener Aufrufmodi:

```
parse(S,E) :- expr(E,S,"").  
  
pretty_print(E,S) :- expr(E,S,"").  
  
normalize(S,T) :- parse(S,E),pretty_print(E,T).
```

- Tests:

```
?- parse("1+(2*3)",E), R is E.  
E = 1+2*3, R = 7.  
  
?- pretty_print(1+2*3,S), !, writef("%s",[S]).  
1+2*3  
  
?- normalize("1+(2*3)",S), !, writef("%s",[S]).  
1+2*3  
  
?- normalize("(1+2)*3",S), !, writef("%s",[S]).  
(1+2)*3
```


Fallstudie: Parsen arithmetischer Ausdrücke

Etwas Reflexion zur Prolog- vs. Haskell-Lösung:

- konzeptionell: entspricht Backtracking entspricht extra „Argument“

```
type Parser a = String → [ (a, String) ]
```

entspricht „Differenzlisten“

- pragmatisch, notationell:

```
term (* (F, T) ) --> factor (F) , "*" , term (T) .  
term (F)         --> factor (F) .
```

vs.

```
term = (factor ++> \f → char '*' +++  
        term ++>  
        \t → yield (Mul f t))  
      ||| factor
```

oder

```
term = do f ← factor  
        char '*'  
        t ← term  
        return (Mul f t)  
      ||| factor
```

oder

```
term = ( Mul <$> factor <*> char '*' <*> term ) ||| factor
```

Deskriptive Programmierung

Diverse andere Spracherweiterungen von Prolog

Zur Erinnerung: Transitive Hülle, aber jetzt mal mit Zyklen

```
direct(frankfurt,san_francisco) .
direct(frankfurt,chicago) .
direct(san_francisco,honolulu) .
direct(honolulu,maui) .
direct(honolulu,san_francisco) .

connection(X, Y) :- direct(X, Y) .
connection(X, Y) :- direct(X, Z), connection(Z, Y) .
```

```
?- connection(san_francisco,Y) .
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
Y = honolulu ;
Y = maui ; ...
```

Ziel sollte sein: Endlossuche vermeiden

Zur Erinnerung: Transitive Hülle, aber jetzt mal mit Zyklen

Idee: schon bereiste Zwischenstationen merken, zum Beispiel als Liste:

```
direct(frankfurt,san_francisco).  
...  
direct(honolulu,san_francisco).  
  
connection(X, Y) :- connection1(X, Y, [X]).  
  
connection1(X, Y, _) :- direct(X, Y).  
connection1(X, Y, L) :- direct(X, Z), not(member(Z,L)),  
                           connection1(Z, Y, [Z|L]).
```

```
?- connection(san_francisco,Y).  
Y = honolulu ;  
Y = maui ;  
Y = san_francisco ;  
false.
```

Eventuell problematisch: lineare Suche in der Zwischenstationsliste.

Alternative: Speichern der besuchten Stationen als Prolog-Fakten.

```
direct(frankfurt,san_francisco).  
...  
direct(honolulu,san_francisco).  
  
connection(X, Y) :- assert(visited(X)), connection2(X, Y).  
  
connection2(X, Y) :- direct(X, Y).  
connection2(X, Y) :- direct(X, Z), not(visited(Z)),  
                           assert(visited(Z)), connection2(Z, Y).
```

```
?- connection(san_francisco,Y).  
Y = honolulu ;  
Y = maui ;  
Y = san_francisco ;  
false.  
  
?- connection(san_francisco,Y).  
Y = honolulu ;  
false.
```

Oops!

„Aufräumen“:

```
direct(frankfurt,san_francisco).  
...  
direct(honolulu,san_francisco).  
  
connection(X, Y) :- retractall(visited(_)),  
                    assert(visited(X)), connection2(X, Y).  
  
connection2(X, Y) :- direct(X, Y).  
connection2(X, Y) :- direct(X, Z), not(visited(Z)),  
                    assert(visited(Z)), connection2(Z, Y).
```

```
?- connection(san_francisco,Y).  
Y = honolulu ;  
Y = maui ;  
Y = san_francisco ;  
false.  
  
?- connection(san_francisco,Y).  
Y = honolulu ;  
Y = maui ;  
Y = san_francisco ;  
false.
```

Beispielverwendungen der Metaprädikate **assert** und **retract**:

```
1 ?- listing.  
true.  
  
2 ?- assert(p(1)).  
true.  
  
3 ?- assert(p(1)).  
true.  
  
4 ?- assert(p(2)).  
true.  
  
5 ?- listing.  
  
:- dynamic p/1.  
p(1).  
p(1).  
p(2).  
true.
```

```
6 ?- p(X).  
X = 1 ;  
X = 1 ;  
X = 2.  
  
7 ?- retract(p(1)).  
true.  
  
8 ?- p(X).  
X = 1 ;  
X = 2.  
  
9 ?- retract(p(X)).  
X = 1 ;  
X = 2.  
  
10 ?- listing.  
  
:- dynamic p/1.  
true.
```

Fakten als Datenstruktur

- Eine nützliche Verwendung von **assert** ist Memoisierung.
- Zur Erinnerung, in Haskell (unmemoisiert):

```
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```



```
fib(N,1) :- N<2, !.
fib(N,M) :- N1 is N-1, fib(N1,M1), N2 is N-2, fib(N2,M2), M is M1+M2.
```

- Das Problem:

```
?- fib(10,X).
X = 89.

?- fib(30,X).
X = 1346269.

?- fib(50,X).
```

hoffnungslos

Fakten als Datenstruktur

```
fib(N,1) :- N<2, !.  
fib(N,M) :- N1 is N-1, fib(N1,M1), N2 is N-2, fib(N2,M2), M is M1+M2.
```



```
:- dynamic(memo/2).  
  
fib(N,1) :- N<2, !.  
fib(N,M) :- memo(N,M), !.  
fib(N,M) :- N1 is N-1, fib(N1,M1), N2 is N-2, fib(N2,M2), M is M1+M2,  
            assert(memo(N,M)).
```

- Nun:

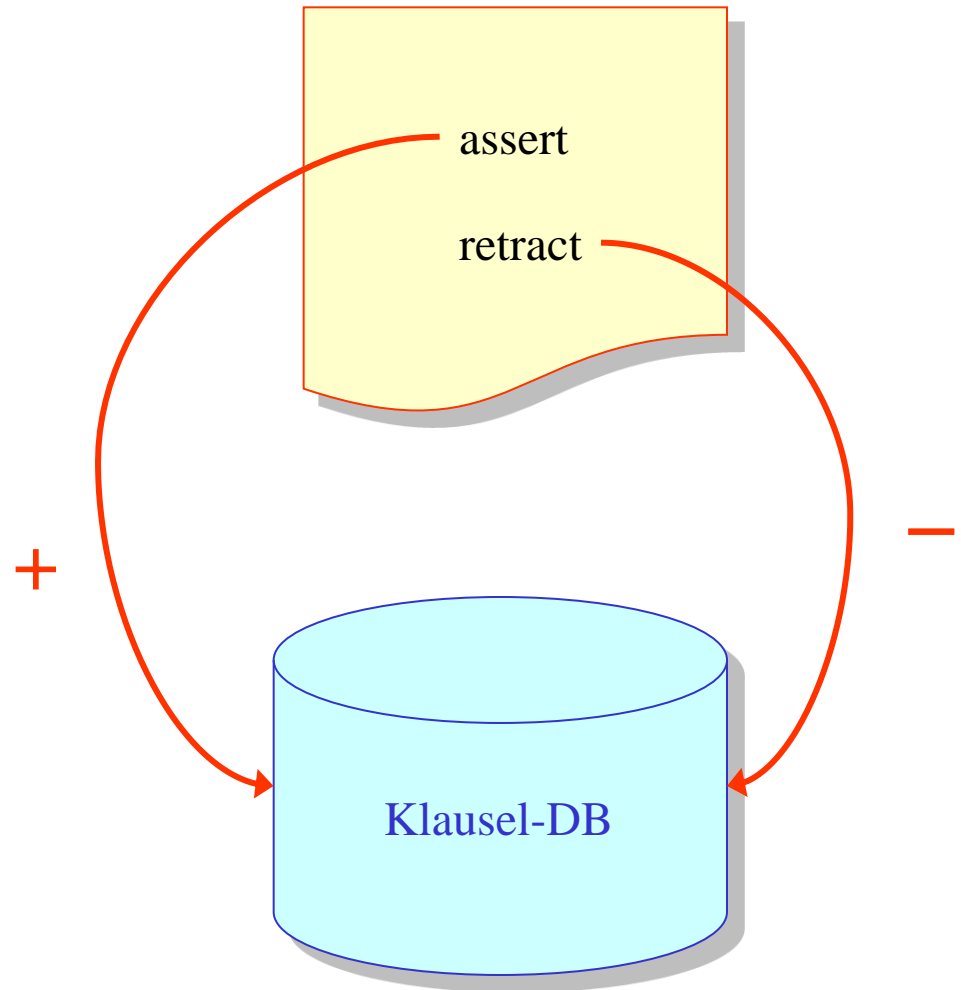
```
?- fib(10,X).  
X = 89.  
  
?- fib(30,X).  
X = 1346269.  
  
?- fib(50,X).  
X = 20365011074.
```

instantan

Seiteneffekte auf die
„Datenbank“ von Klauseln!

zwei Varianten üblich:

- 1) „DB“ als zusätzliche
Datenstruktur (Fakten)
⇒ (fast schon) normal in LP
- 2) Selbstmodifikation des
Programms
(„DB“ als Programm)
⇒ Meta-Programmierung



Generierung aller Lösungen einer Anfrage (1)

- Oft existieren ja mehrere Lösungen zu einer Anfrage:

```
child(martha, charlotte).
child(charlotte, caroline).
child(caroline, laura).
child(laura, rose).

descend(X, Y) :- child(X, Y).
descend(X, Y) :- child(X, Z), descend(Z, Y).
```

Die Anfrage `?- descend(martha, X) .` würde **sukzessive** die Antworten `X = charlotte`, `X = caroline`, `X = laura` sowie `X = rose` liefern.

- Prolog bietet drei verschiedene Meta-Prädikate, um alle Lösungen „auf einen Schlag“ zu generieren:

`findall`, `bagof`, `setof`

und sie auf jeweils eine bestimmte Art in einer Ergebnisliste aufzuführen.

Generierung aller Lösungen einer Anfrage (2)

```
findall(Template, Goal, List).
```

- Für jede Lösung der Anfrage `Goal` wird das instantiierte `Template` in die Ergebnisliste `List` aufgenommen.

```
?- findall(X, descend(martha, X), Z).  
Z = [charlotte, caroline, laura, rose].
```

- Der Term `Template` kann auch eine ganze Struktur mit (oder ohne) Variablen sein, woraus dann die Einträge der Ergebnisliste „gebaut“ werden.

```
?- findall(fromMartha(X), descend(martha, X), Z).  
Z = [fromMartha(charlotte), fromMartha(caroline),  
      fromMartha(laura), fromMartha(rose)].
```

Generierung aller Lösungen einer Anfrage (3)

Variante: `bagof(Template, Goal, List)`.

Die nicht im `Template` vorkommenden freien Variablen werden getrennt gebunden:

```
?- bagof(X, descend(Y, X), Z).  
Y = caroline,  
Z = [laura, rose] ;  
  
Y = charlotte,  
Z = [caroline, laura, rose] ;  
  
Y = laura,  
Z = [rose] ;  
  
Y = martha,  
Z = [charlotte, caroline, laura, rose].
```

Zum Vergleich:

```
?- findall(X, descend(Y, X), Z).  
Z = [charlotte, caroline, laura, rose, caroline, laura, ...].
```

Generierung aller Lösungen einer Anfrage (4)

weitere Variante: `setof(Template, Goal, List).`

... verhält sich wie `bagof`, allerdings werden Duplikate aus der Ergebnisliste gelöscht, und die Liste sortiert.

Denkbare Verwendung der „Collection“-Prädikate: Simulation von list comprehensions.

Haskell:

```
[ e | x ← xs ]
```



Prolog:

```
findall(E, member(X,Xs), List).
```

Generierung aller Lösungen einer Anfrage (5)

Beispiele:

Prolog-Äquivalente zu folgenden Haskell-Definitionen?

1.

```
[ n .. m ]
```

2.

```
[ n, m .. 1 ]
```

3.

```
[ x * x | x ← [1 .. 100], x `mod` 2 == 0 ]
```

Mögliche Lösungen zu 1.:

```
fromTo(N,M,L) :- N > M, !, L = [].  
fromTo(N,M,[N|L]) :- N1 is N+1, fromTo(N1,M,L).
```

oder

```
fromTo(N,M,L) :- forall(X,between(N,M,X),L).
```

Generierung aller Lösungen einer Anfrage (6)

Beispiele:

Prolog-Äquivalente zu folgenden Haskell-Definitionen?

2.

```
[ n, m .. 1 ]
```

3.

```
[ x * x | x ← [1 .. 100], x `mod` 2 == 0 ]
```

Mögliche Lösungen zu 2.:

(Haskell „erlaubt“ übrigens auch
[0, 0 .. 5] und [0, -2 .. -5].)

```
fromThenTo (N,M,L,Xs)      :- (N >= M; N > L), !, Xs = [].  
fromThenTo (N,M,L,[N|R]) :- M1 is M+M-N, fromThenTo (M,M1,L,R) .
```

oder

```
fromThenTo (N,M,L,Xs) :- (N >= M; N > L), !, Xs = [].  
fromThenTo (N,M,L,Xs) :- D is M-N, fromTo (0, (L-N)/D, Is),  
                          findall(X, (member(I,Is), X is N+I*D), Xs) .
```


Generierung aller Lösungen einer Anfrage (7)

Beispiele:

Prolog-Äquivalente zu folgenden Haskell-Definitionen?

3. `[x * x | x ← [1 .. 100], x `mod` 2 == 0]`

Mögliche Lösungen zu 3.:

```
squares(L) :- fromTo(1,100,Xs), filter(Xs,Ys), map(Ys,L).  
  
filter([],[]).  
filter([X|Xs],[X|Ys]) :- X mod 2 == 0, !, filter(Xs,Ys).  
filter([_ |Xs],Ys)      :- filter(Xs,Ys).  
  
map([],[]).  
map([X|Xs],[Y|Ys]) :- Y is X*X, map(Xs,Ys).
```

oder

```
squares(L) :- fromTo(1,100,Xs),  
              findall(Y, (member(X,Xs), X mod 2 == 0, Y is X*X),L).
```

Deskriptive Programmierung

FP vs. LP (oder doch nicht „vs.“?)

FP vs. LP: einige prinzipielle Entsprechungen

funktional (Haskell)

logisch (Prolog)

Funktion

Relation

Gleichung

Klausel

Schachtelung v. Ausdrücken

Konjunktion v. Literalen

Reduktion

Resolution

Pattern Matching

Unifikation

lazy evaluation (leftmost-outermost)

sequentielle Abarbeitung (left-right)

list comprehensions

findall/bagof/setof-Literale

Parserkombinatoren

Definite Clause Grammars

FP vs. LP: einige prinzipielle Abweichungen

funktional (Haskell)

logisch (Prolog)

???

freie Variablen, Aufrufmodi

???

Lösungsalternativen

???

Backtracking

???

Cut

???

Negation

Typen, Polymorphie

???

Higher-Order

???

mathematische Purheit

(nur bedingt)

Zum Beispiel in der Sprache Curry:

```
coin :: Int
coin = 0
coin = 1
```

```
double :: Int → Int
double x = x + x
```

```
> coin
0
More?
1
More?
No more Solutions
```

```
> double coin
0
More?
2
More?
No more Solutions
```

```
coin :: Int
coin = 0 ? 1
```

Zum Beispiel in der Sprache **Curry**:

```
f :: a → [a] → [a]
f x ys      = x : ys
f x (y : ys) = y : f x ys
```

```
g :: [a] → [a]
g []      = []
g (x : xs) = f x (g xs)
```

```
> f 3 [1,2]
[1,2,3]
More?
[1,3,2]
More?
[3,1,2]
More?
No more Solutions
```

```
> g [1,2,3]
[3,2,1]
More?
[3,1,2]
More?
[2,3,1]
More?
...
```

Zum Beispiel in der Sprache Curry:

```
list :: [Int]
list = ys ++ [1]
  where ys free
```

```
f :: [a] → a
f xs | ys ++ [y] == xs = y
  where ys, y free
```

```
> list
[1]
More?
[_a,1]
More?
[_a,_b,1]
More?
...
```

```
> f [1..4]
4
More?
No more Solutions
```

```
f :: [a] → a
f (_ ++ [y]) = y
```

Zebra-Puzzle funktional-logisch (1)

```
data Color      = Red | Yellow | Blue | Green | Ivory
data Nationality = Norwegian | Englishman | Spaniard | Ukrainian | Japanese
data Drink      = Coffee | Tea | Milk | Juice | Water
data Pet        = Dog | Horse | Snails | Fox | Zebra
data Smoke      = Winston | Kools | Chesterfield | Lucky | Parliaments
```

```
right_of :: a → a → [a] → Success
```

```
right_of r l (h1 : h2 : hs) = (l == h1 & r == h2) ? right_of r l (h2 : hs)
```

```
next_to :: a → a → [a] → Success
```

```
next_to x y = right_of x y
```

```
next_to x y = right_of y x
```

```
member :: a → [a] → Success
```

```
member x (y : ys) = x == y ? member x ys
```


Zebra-Puzzle funktional-logisch (2)

```
zebra :: [(Color,Nationality,Drink,Pet,Smoke)], Nationality)
zebra | member (Red, Englishman, _, _, _) houses
      & member (_, Spaniard, _, Dog, _) houses
      & member (Green, _, Coffee, _, _) houses
      & member (_, Ukrainian, Tea, _, _) houses
      & right_of (Green, _, _, _) (Ivory, _, _, _) houses
      & member (_, _, _, Snails, Winston) houses
      & member (Yellow, _, _, _, Kools) houses
      & next_to (_, _, _, _, Chesterfield) (_, _, _, Fox, _) houses
      & next_to (_, _, _, _, Kools) (_, _, _, Horse, _) houses
      & member (_, _, Juice, _, Lucky) houses
      & member (_, Japanese, _, _, Parliaments) houses
      & next_to (_, Norwegian, _, _, _) (Blue, _, _, _, _) houses
      & member (_, zebraOwner, _, Zebra, _) houses
      & member (_, _, Water, _, _) houses
= (houses, zebraOwner)
where
  houses = [(_, Norwegian, _, _, _), _, (_, _, Milk, _, _), _, _]
  zebraOwner = _
```

Ende