K09

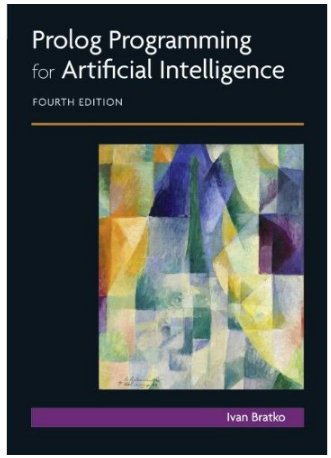
Prolog

Programmierung in Logik

Das vielleicht überraschendste Programmierparadigma, dass aber gleichzeitig am wenigsten verbreitet ist.

teilweise basiert auf Folien von Kristina Striegnitz

Literatur

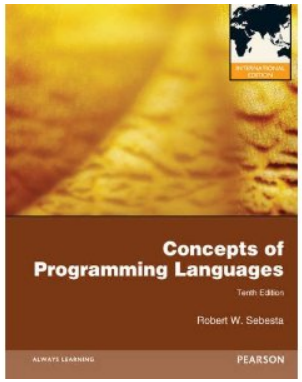


- Ivan Bratko: Prolog Programming for Artificial Intelligence. 4th Edition. Person, 2011. ISBN: 978-0321417466

Frei/Online verfügbar:

- P. Blackburn, J. Bos, K. Striegnitz: Learn Prolog Now! College Publications, 2006. ISBN: 978-1904987178 http://www.learnprolognow.org/index.php

Kapitel zu Prolog in eingangs erwähnten Büchern:



Herkunft

- Entstammt dem automatischen und deduktiven beweisen von Theoremen in der Logik
  - Deduktion (lat. Ableitung):

    Schliessen vom Allgemeinen auf das Besondere

    - Beispiel: **Modus Ponens** (Abtrennungsregel) – bekannt seit ARISTOTELES

p	(Prämisse 1)
p → q	(Prämisse 2)
q	(Konklusion)

Es regnet. — Fakt

Wenn es regnet dann ist die Strasse nass. — allgemeingültige Prinzip

Die Strasse ist nass. — spezifische Schlussfolgerung

  - Prolog basiert insbesondere auf der Prädikatenlogik (First Order Logic).

Historische Entwicklung

- Prolog als Umsetzung des „Computation as Deduction“ Paradigma wurde Anfang der 1970er Jahre entwickelt.

19. Jhd.	G. FREGE und C.S. PEIRCE entwickeln unabhängig die Prädikatenlogik
1930er Jahre	K. GÖDEL und J. HERBRAND schaffen die theoretischen Grundlagen zum „Computation as Deduction“
1965	A. ROBINSON: Resulotion und Unifikation als Algorithmen zum automischen deduktiven Schliessen
1971-1973	A. COLMERAUER, et al. entwickeln Prolog und den ersten Interpreter
1982-1991	Prolog-Variante als Basis des „Japanese Fifth Generation Project“ (Etat ca. 500Mio $); Ziel „epoch-making computer“; Wird als Fehlschlag oder als seiner Zeit voraus eingeschätzt
1995	Edingburgh-Dialekt wird zum ISO-Standard (ISO/IEC 13211-1)

Anwendungsgebiete

- Liegen weitestgehend im Gebiet der Künstlichen Intelligenz und Wissensverarbeitung:
  - Computerlinguistik; z.B.: innerhalb IBM Watson benutzt
  - Deduktive (relationale) Datenbanken
  - Expertensysteme
  - Kombinatorische Optimierung

- Prolog hat später auch die Entwicklung neuer und logikbasierter Problemlösungsansätze beeinflusst:
  - Constraint Logic Programming: Programm ist eine Menge von Randbedingungen (Constraints) für die eine Lösung zu finden ist, die alle Randbedingungen erfüllt.
  - Inductive Logic Programming: Logikbasierter Ansatz zum Maschinellen Lernen.

Gegenüberstellung

Gesucht: Programm dass alle dreistelligen Zahlen berechnet die durch 5 und durch 6 teilbar sind.

```
int main() {                          ///// C++ /////
  for (int i=100; i<999; i++) {
    if ((i%5==0) && (i%6==0)) cout<<i<<" ";
  }
}
```

```
----- Haskell -----
main = return [x | x <- [100..999], mod x 5 == 0, mod x 6 == 0]
```

```
%%%%% Prolog %%%%%
ziffer(Z)      :- member(Z,[0,1,2,3,4,5,6,7,8,9]).
teilbar(Zahl) :- Zahl is H*100 + Z*10 + E,
                 ziffer(H), ziffer(Z), ziffer(E),
                 H > 0,
                 0 is Zahl mod 5,
                 0 is Zahl mod 6.
```

Gegenüberstellung

- C++:
  - Prozedural: Beschreibung einer Abfolge von Anweisungen die zusammen beschreiben wie das Problem zu lösen ist.
- Haskell:
  - Deklarativ: Beschreibung der Eigenschaften der Lösung, nämlich die Liste der Zahlen zwischen 100 und 999, so dass jede Zahl durch 5 und 6 teilbar ist.
- Prolog:
  - Deklarativ: Beschreibung des Problems bzw. der Kriterien die erfüllt sein müssen:
    1. Was ist eine Ziffer?
    2. Welche Eigenschaften haben dreistellige und durch 5 und 6 teilbare Zahlen?

Doppelte Nutzbarkeit

- Dasselbe Prolog Programm erfüllt zweierlei Zwecke:
  1. Anfrageauswertung – Testing
  2. Werteberechnung – Computing

```
?- teilbar(100).
false.

?- teilbar(120).
true.

?- teilbar(X).
X = 120 ;
X = 150 ;
X = 180 ;
```

Ist 100 eine dreist. und durch 5 u. 6 teilbare Zahl?

Antwort: Nein.

Ist 120 eine dreist. und durch 5 u. 6 teilbare Zahl?

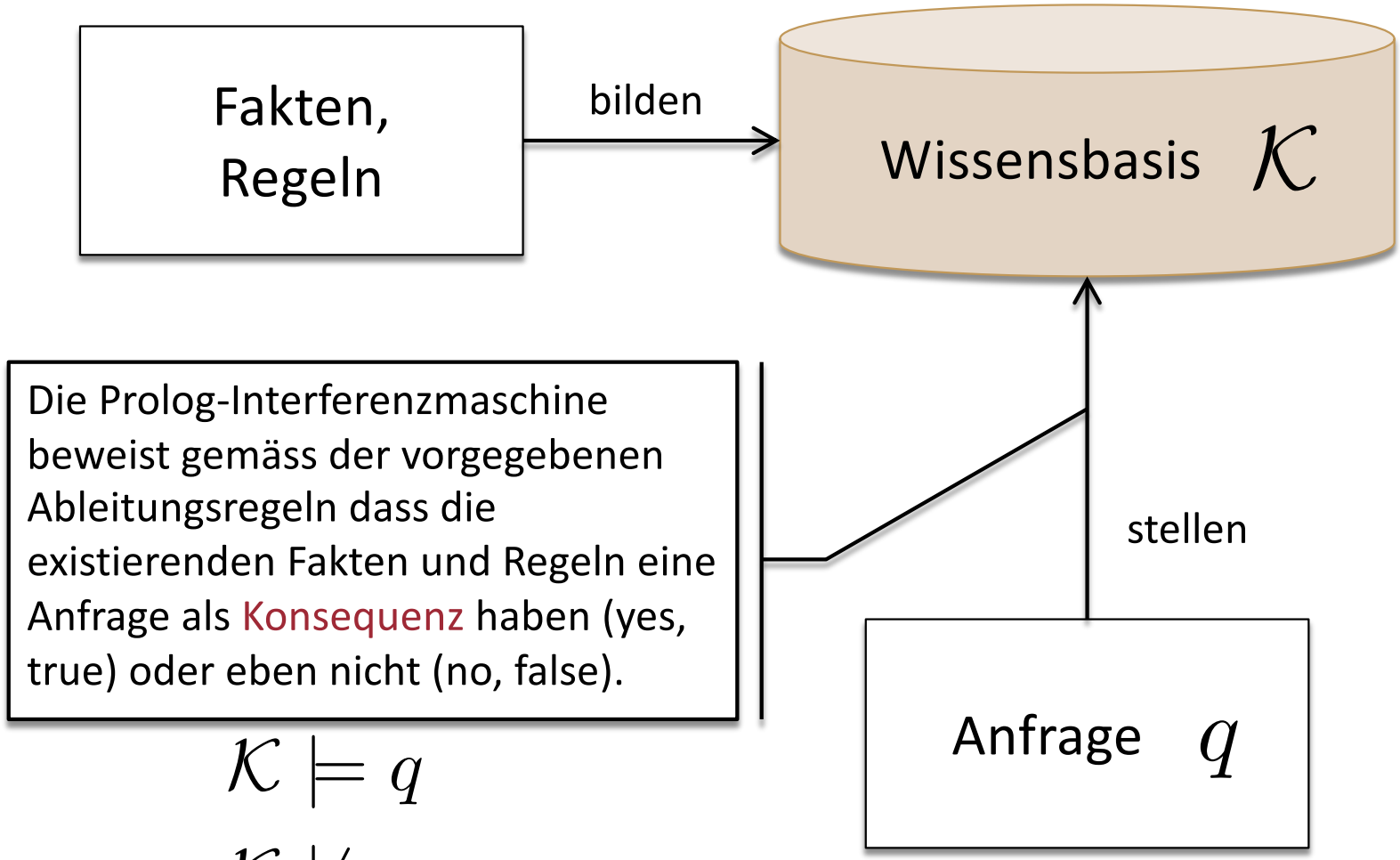
Antwort: Ja.

Finde einen Wert für **`X`** der **`teilbar`** erfüllt!

Erster gefundener Wert: 120; zweiter: 150; usw.

Frage: Wie findet Prolog die Antworten bzw. welche Algorithmen benutzt Prolog dazu?

Das Prinzip von Prolog

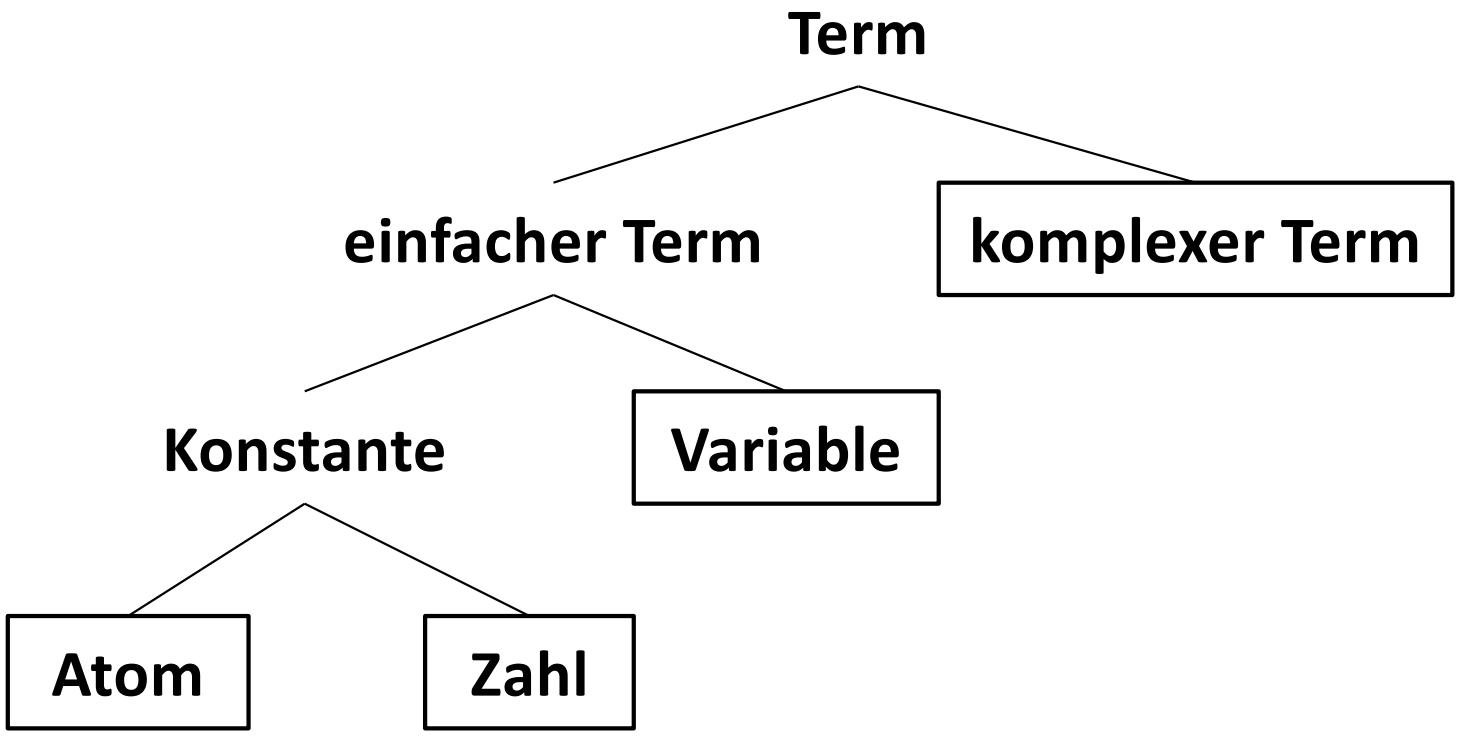


$$\mathcal{K} \models q$$

$$\mathcal{K} \not\models q$$

Syntax: Terme

- Ein Term* ist der einzige Datentyp in Prolog
  - Im Grunde genommen ein algebraischer Datentyp



* Entspricht einem Term in der Prädikatenlogik, obwohl die genaue Syntax nicht identisch ist.

Atome & Zahlen

- Atome sind Namen (Symbole) die Objekte und Relationen bezeichnen.
  - Anforderungen:
    - Folge von Buchstaben, Ziffern und diversen Sonderzeichen +-*<>&#@:
    - Beginnen mit einem Kleinbuchstabe, es sei denn es sind Zeichenketten '...'

  Beispiele: **`ziffer`**, **`teilbar`**, **`alice`**, **`prolog`**, **`kennt`**, **`'ABC'`**, **`+`**, **`:-`**

- Zahlen sind entweder Fest- oder Gleitkommazahlen
  - Wertebereich ist u.U. Maschinenabhängig (32, 64 bit)

  Beispiele: **`1`**, **`12345`**, **`-4`**, **`3.14`**, **`-1.11`**, **`-3.9E+7`**, **`1.8E-6`**, `2.0E4` ✗

Variablen

- Eine Variable steht für ein Objekt dass noch unbekannt ist und für die Prolog eine Lösung finden muss.
  - Syntaktische Anforderungen:
    - Folge von Buchstaben, Zahlen und diversen Sonderzeichen
    - Beginnen mit einem Grossbuchstabe oder **_**
  - Spezialfall: Anonyme Variable **_**
    - Wird benutzt in Anfragen wenn der Wert nicht von Interesse in der Lösung ist.

Beispiele: **`X`**, **`Y`**, **`_variable`**, **_**

Komplexe Terme

- Atom*, gefolgt von ein- oder mehreren Termen die in runden Klammern stehen und durch Komma getrennt sind.

Beispiele:

```
vaterVon(homer, bart)
mutterVon(marge, X)
zeichentrickserie(simpsons)
bearbeite(gegenstand, W, zustand(aktZustand))
```

Stelligkeit (engl. aritiy): Anzahl der Argumente

```
vaterVon/2
mutterVon/2
zeichentrickserie/1
homer/0
h(X, i(i(i(b))))
```

Arität?

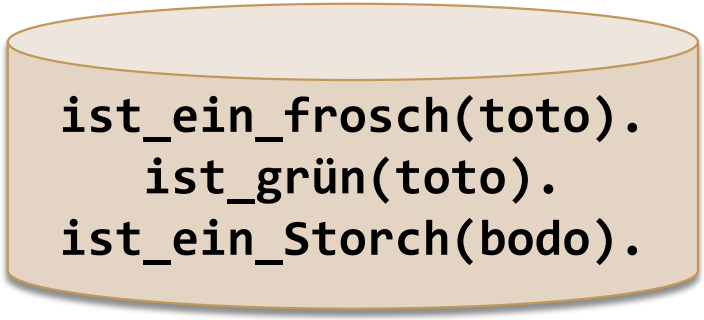
* Auch Funktor genannt; wobei Funktor hier in seiner Bedeutung in der Logik zu verstehen ist.

Wissensbasen (i)

- Prolog Programme sind Wissensbasen; genauer gesagt eine Menge von:
  - Fakten: Aussagen die per Definition gelten (Wahrheitswert = true)
  - Regeln: Drücken Implikationen (wenn dann) zwischen Fakten aus.

  Klauseln

Beispiel (Wissensbasis die nur Fakten enthält):



Aussagen:
Toto ist ein Frosch.
Toto ist grün.
Bodo ist ein Storch.

Andere Möglichkeiten:

```
frosch(toto).        |  f(toto).        |  istTier(frosch, toto).
grün(toto).          |  g(toto).        |  hatFarbe(grün, toto).
storch(bodo).        |  s(bodo).        |  istTier(storch, bodo).
```

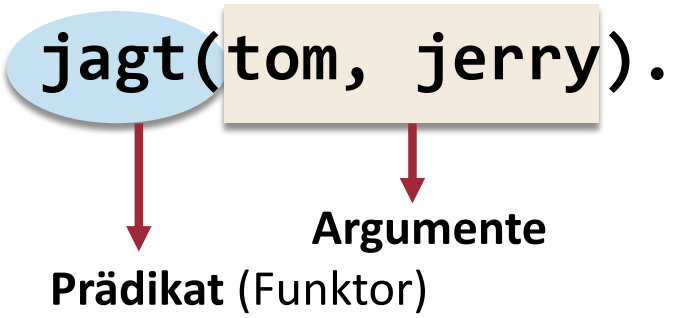
Wissensbasen (ii)

Beispiel (Wissensbasis mit Fakten und Regeln):

```
listensToMusic(mia).
happy(yolanda).
happy(X) :- listensToMusic(X).
playsInstrument(X) :- playsGuitar(X).
musician(X) :- playsInstrument(X), performsConcert(X).
```

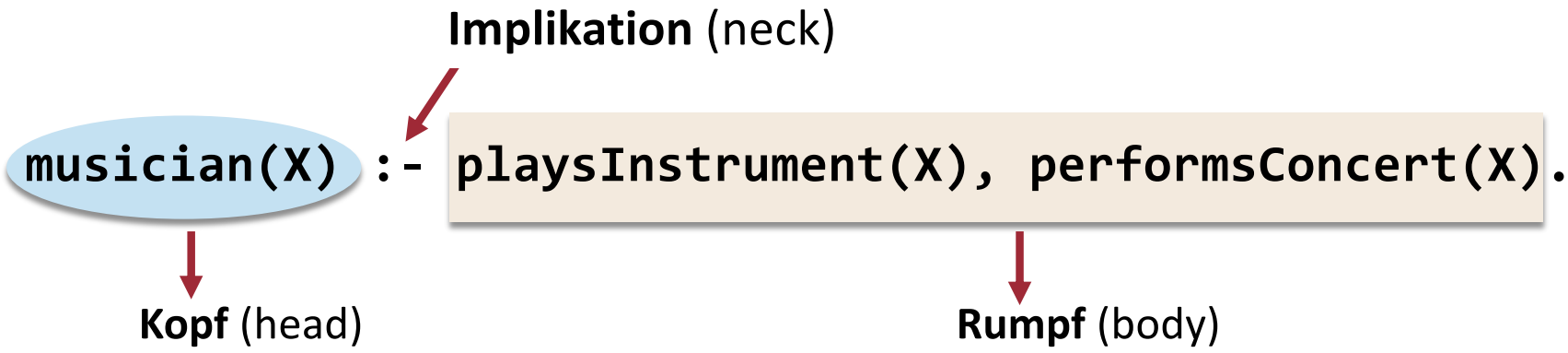
Diese Wissensbasis enthält zwei Fakten und drei Regeln, bzw. fünf Klauseln.

Fakten



- Enden mit einem Punkt.
- Haben beliebig viele Argumente.
  - In Fakten sind Variablen allquantifiziert: `∀X: jagt(X, jerry) !`
  - In Anfragen sind Variablen existenzquantifiziert: `∃X: jagt(X, jerry) ?`
- Prädikat beginnt mit einem Kleinbuchstabe.

Regeln



- Gültigkeitsbereich (Scope) einer Variable erstreckt sich nur auf Regel selbst:

```
h1(X,Y) :- a(X), b(Y), c(X).
h2(X,Y) :- a(Y), d(X).
```

- Für einen Kopf kann es mehrere Regeln geben; realisiert ein Oder:

```
h(X) :- a(X).
h(X) :- b(X).
```

- Beachte: ein Fakt ist eine Regel ohne Body:

  `katze(tom).` ist äquivalent zu `katze(tom) :- true.`

Anfragen und Ziele

- Anfrage:
  - Sequenz von einem oder mehreren Zielen
    - Jedes Ziel ist ein Term.
    - Ziele durch Komma getrennt; realisiert ein Und.
  - Boolesche Anfrage
    - Hat keine Variablen und kann demzufolge nur ...
    - ... mit **`true`** (**`yes`**) oder **`false`** (**`no`**) beantwortet werden.
  - Ergänzungs- oder Ergebnisanfrage
    - Enthält mindestens eine Variable.
    - Prolog-Inferenzmaschine versucht eine Variablenbelegung zu finden.
    - Unter Umständen existieren unendlich viele verschiedene Variablenbelegungen.

Lösungssuche

- Allgemein:

  Ist *q* eine Ergänzungsanfrage und
  f ein Fakt (einer Wissensbasis),

  so sucht Prolog nach einer Variablenbelegung,

  so dass *q* == *f* gilt (*q* und *f* sind identisch).

- Dazu werden zwei Algorithmen verwendet:
  - Unifikation
  - Backtracking mit optionalem Cut

Unifikation (Gleichmachen)

Definition: zwei Terme **s**, **t** sind unifizierbar, d.h. **s = t**, wenn:

1. Sind **s** und **t** Konstanten, dann muss gelten **s == t**.
2. Entweder **s** oder **t** eine Variable ist, wobei dabei entweder **s** mit **t** instanziiert wird, oder umgekehrt.
3. Sind **s** und **t** komplexe Terme **s: $f(s_1, \ldots, s_n)$**, **t: $g(t_1, \ldots, t_m)$**, dann muss gelten:
   1. **f == g**
   2. **n == m**
   3. **$s_i = t_i$** für alle **i = 1, ... n**.

Beispiele (nicht)unifizierbarer Terme

nur Konstanten

```
?- mia = mia.     % Infixschreibweise
true.
?- =(mia, mia).   % Präfixschreibweise
true.
?- 'Mia' = mia.   % Prolog ist case-sensitive
false.
?- 'mia' = mia.   % keine Typunterscheidung
true.
?- mia = vincent.
false.
```

mit Variablen

```
?- mia = X.
X = mia
?- X = father(butch).
X = father(butch)
?- X = Y.
X=Y
```

```
?- X = Y, X = mia.
X = mia, Y = mia
?- X=Y, X=m, Y=v.
false.
```

Beispiele (nicht)unifizierbarer Terme

komplexe Terme

```
?- kill(shoot(gun),Y) = kill(X,stab(knife)).
X = shoot(gun),
Y = stab(knife)

?- kill(shoot(gun), stab(knife)) = kill(X,stab(Y)).
X = shoot(gun),
Y = knife

?- loves(X,X) = loves(marcellus,mia).
false.
```

zyklische Terme

```
?- father(X) = X.
X = father(X).
```

Ältere Prolog-Implementierungen liefen hier in Endlosschleife.

Abarbeitungsreihenfolge

- Anfragen werden von links nach rechts abgearbeitet.
- Wissensbasis wird dabei von oben nach unten durchsucht.
- Solange, bis alle Ziele erfüllt sind, bzw. bis nichts mehr unifiziert werden kann aber noch nicht alle Ziele erfüllt sind.

→ Ergo: Reihenfolge der Ziele bzw. Fakten kann entscheidend sein.

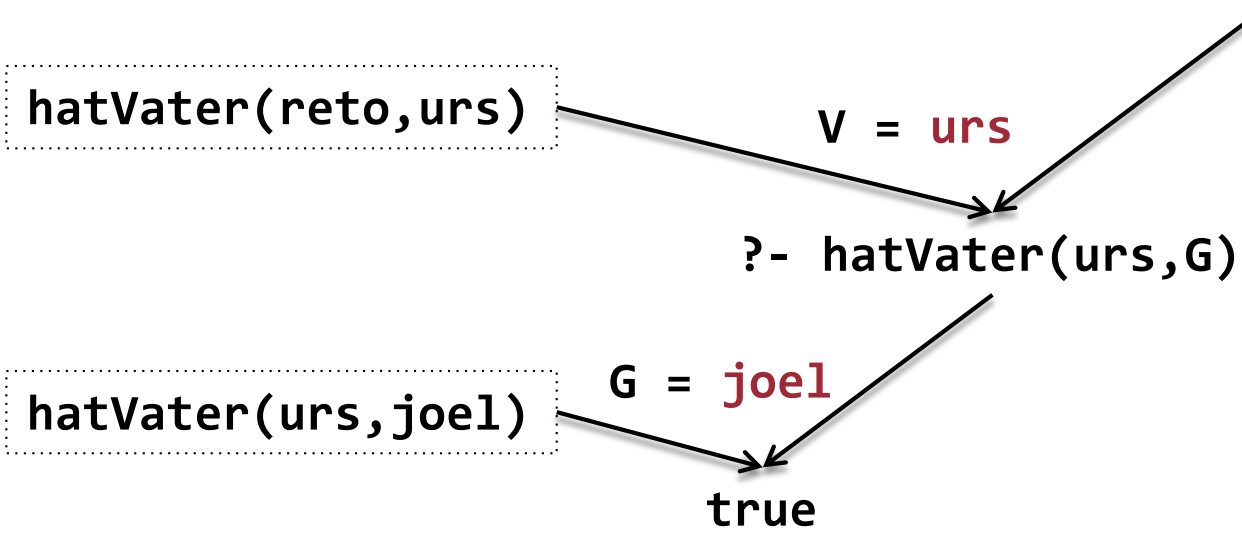
Beispiel Abarbeitungsreihenfolge

Wissensbasis mit folgenden Fakten:

```
hatVater(jana, urs).
hatVater(reto, urs).
hatVater(urs, joel).
hatPartner(urs, eva).
hatPartner(joel, emma).
```

Anfrage: Wer ist der Vater bzw. Grossvater von Reto?

```
?- hatVater(reto,V), hatVater(V,G)
```



Backtracking

- Idee: Wenn ...
  - ... klar ist dass eine Teillösung nicht zu einer endgültigen Lösung führen kann (Sackgasse), oder ...
  - ... wenn klar ist dass nachdem eine Lösung gefunden wurde noch weitere alternative Wege existieren, dann ...
  - ... gehe ein oder mehrere Schritte zurück und suche in den noch nicht verfolgten alternativen Wegen nach Lösungen.

  - Dies stellt sicher dass schlussendlich alle in Frage kommenden Wege, egal ob sie zu einer Lösung führen, verfolgt wurden.

Beispiel Backtracking

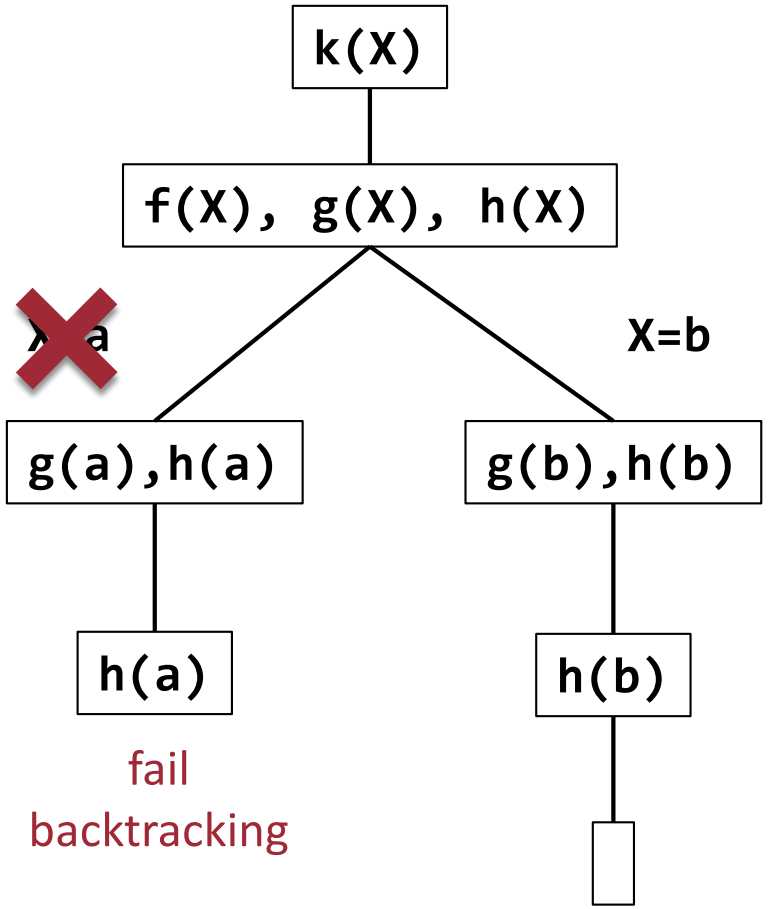
Fakten in Wissensbasis:

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):-f(X),g(X),h(X).
```

Anfrage:

```
?- k(X).
```

Suchbaum:



Backtracking – Cut (i)

- „Ungezügeltes“ Backtracking kann zu zweierlei Problemen führen:
  - Erzeugung inkorrekter Lösungen.

```
% groesser(X, Y, M) – M ist grössere Zahl von X und Y
groesser(X,Y,X) :- X > Y.
groesser(X,Y,Y).

?- groesser(2,1,X).
X = 2 ;    % und jetzt Backtracking
X = 1.     % uups; Match der zweiten Klausel
```

  - Weiterverfolgung alternativer Pfade obwohl dies nicht notwendig ist.

```
% max(X, Y, M) – M ist Maximum von X und Y
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y. % Muss nicht geprüft
                     % werden wenn X >= Y.
```

Backtracking – Cut (i)

- Das Prädikat **!** (Cut) unterbindet Backtracking.
  - Steht als Ziel in einer Regel.
  - Ist als Ziel selbst immer erfüllt.

```
% groesser(X, Y, M) – M ist grössere Zahl von X und Y
groesser(X,Y,X) :- X > Y, !.
groesser(X,Y,Y).

?- groesser(2,1,X).
X = 2.    % kein Backtracking → keine weitere Lösung
```

```
% max(X, Y, M) – M ist Maximum von X und Y
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y) :- X < Y. % Wird nicht geprüft wenn X >= Y.
```

Arithmetik

- Arithmetische Zuweisung erfolgt über das eingebaute **`is`** Prädikat:

  `X is 3*4.` oder **`is(X,3*4).`**

  - Wenn X ungebunden, dann wird Ausdruck ausgewertet und das Ergebnis X zugewiesen:
    - **`X is 1+2.`** (**`X=3`**)
    - **`X is sqrt(9)`** (**`X=3`**)
    - **`X is X.`** (**`fail`**)
  - Wenn X gebunden, dann wird der Ausdruck ausgewertet und das Ergebnis mit X verglichen:
    - **`X is 1, X is 2.`** (**`false`**)
    - **`X is 1, X is X+1.`** (**`false`**)
    - **`X is 1, Y is 1, X is Y.`** (**`true`**)
    - **`X is 2+2, X is 1+3.`** (**`true`**)

Arithmetische Operatoren

- Auswahl binärer Operatoren:

  `+`, `-`, `*`, `/`, `mod`, `**`

- Logische Operatoren:

`,`	(und)
`;`	(oder)
`not`	(nicht)

- Vergleiche:

`=` und `\=`	Test ob Unifikation (nicht) möglich ist.
`==` und `\==`	Test ob Terme (nicht) identisch sind.
`=:=` und `=\=`	Test ob Ausdrücke nach Auswertung (un)gleich sind. Analog für: `<`, `>`, `=<`, `>=`

Gegenüberstellung Vergleiche

- `=` und `==` (Unifikation versus Identität):

```
?- X = a.                    ?- X == a.
X = a.                       false.
?- X = Y.                    ?- X == Y.
X = Y.                       false.
                             ?- X = Y, X == Y.
                             true.
```

- `==` und `=:=` (Identität versus Gleichheit nach Auswertung):

```
?- 1+2 == 1+2.               ?- 1+2 =:= 1+2.
true.                        true.
?- 1+2 == 2+1.               ?- 1+2 =:= 2+1.
false.                       true.
```

Rekursion

- Auch in Prolog existieren (wie in Haskell) keine Kontrollstrukturen.
- Rekursion lässt sich in Prolog meist über zwei Klauseln mit demselben Prädikat realisieren:
  - Erste Klausel liefert Abbruchbedingung.
  - Zweite Klausel kodiert Rekursionsschritt.

Beispiele:

```
vorfahrVon(X,Y) :- elternteilVon(X,Y).
vorfahrVon(X,Y) :- elternteilVon(X,Z), vorfahrVon(Z,Y).

fak(0,1).
fak(N,X) :- N>0, M is N-1, fak(M,Y), X is N*Y.
```

Listen

- Werden (wie in Haskell) durch eckige Klammern geschrieben:

  `[1,2,3]`

  äquivalent zu **`.(1,.(2,.(3,[])))`** (**`.`** ist „cons“ Operator)

- Können unterschiedliche Terme enthalten:

  `[1,foo,pred(bar),[2,joe]]`

- Zugriff auf Elemente über Unifikation und **`|`** Operator:

  `[Head|Tail] = [1,2,3].` (**`Head=1`**, **`Tail=[2,3]`**)

  `[[X,Y]|Z] = [[1,2],[3]].` (**`X=1`**, **`Y=2`**, **`Z=[3]`**)

- Verschiedene eingebaute Prädikate zur Listenmanipulation:

  `is_list`, **`member`**, **`append`**, **`sublist`**,
  `delete`, **`nth0`**, **`reverse`**, **`length`**