# Künstliche Intelligenz

Prof. Dr. Dirk Krechel
Hochschule RheinMain

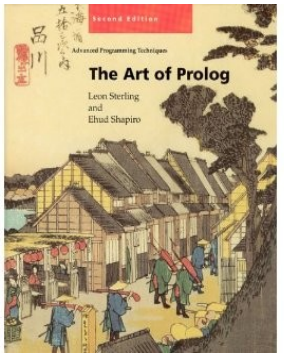
Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim Geisenheim

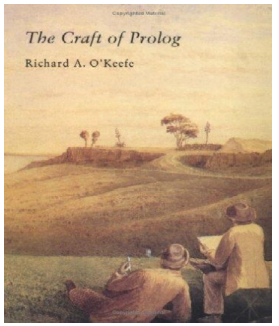
# Inhalte

- Einführung
- **Symbolische Verfahren, Logik**
  - Aussagenlogik, Prädikatenlogik
  - **Horn Logik, Prolog**
- Suchen und Bewerten
  - Problemlösen durch Suche
    - Uninformierte Suche
    - Heuristische Suche
    - Spielbäume
  - Information Retrieval
- Lernen
  - Entscheidungstheorie
  - Naive Bayes
  - Entscheidungsbäume
  - Neuronale Netze
  - unüberwachtes Lernen

# Hornlogik und Prolog

- Hornlogik
  - Prädikatenlogik erster Stufe mit Einschränkungen
    - Klauselnormalform, nur ein positives Literal, weniger ausdrucksmächtig
  - Resolution vollständig, keine Faktorisierung
- Prolog
  - Logisches/Deklaratives Programmieren
  - Basiert auf Horn-Logik und SLD-Resolution
  - Programmiersprache Prolog
- Literatur und Online-Quellen
  - SWI-Prolog, http://www.swi-prolog.org
    - Umfassende Features, ausgereift, plattformunabhängig
  - Prolog-Programmieren – Kunst und Handwerk
    - The Art of Prolog
      Sterling Shapiro, 1994
    - The Craft of Prolog
      O'Keefe, 1990*„Elegance is not optional“*



# Hornlogik

- Hornlogik
  - Echte Untermenge der Prädikatenlogik
  - Formeln in Klauselnormalform
  - Aber nur ein positives Literal!
- Modellierung
  - Implikationen einfach zu realisieren
  - Typisch, bei Abbildung von Wissen
  - Vorwärtsregeln: Wenn das und das und das und .... dann das
  - Aber *echte* Untermenge, manche Aussagen nicht ausdrückbar
- Beispiel
  - {P(x), Q(y)} ist *keine* Hornklausel
  - {¬P(x), Q(y)} ist eine Hornklausel
  - {¬P(x), ¬Q(y), R(y)} ist eine Hornklausel
  - {P(x)} ist eine Hornklausel

$\alpha, \alpha_i, \beta, \beta_i$ sind Atomformeln

$$\neg\alpha_1 \vee \ldots \vee \neg\alpha_n \vee \beta$$

$$\alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow \beta$$

$$\alpha_1 \wedge \alpha_2 \Rightarrow \beta_1 \vee \beta_2$$

$$\neg\alpha_1 \vee \neg\alpha_2 \vee \beta_1 \vee \beta_2$$

# Hornlogik – Fakten, Regeln, Anfragen

- Wir nennen einen Hornklausel
- *Fakt*
  - wenn sie nur aus einer positiven atomaren Formel besteht
  - Beispiel: { P(x) } $\Rightarrow P(x)$
- *Regel*
  - wenn sie aus einer positiven und mindestens einer negativen atomaren Formel besteht
  - Beispiel: $\{\neg P(x), \neg Q(y), R(x)\}$ $P(x) \wedge Q(y) \Rightarrow R(x)$
- *Anfrage*
  - wenn sie ausschließlich aus negativen atomaren Formeln besteht
  - Beispiel: { $\neg P(x), \neg Q(x)$ } $P(x) \wedge Q(x) \Rightarrow$
    Gilt P(x) und Q(x)?
    Kann man P(x) und Q(x) aus Wissensbasis $\Sigma$ folgern? … gdw … $\Sigma \nvdash P(x) \wedge Q(x)$
    Ist $\neg P(x), \neg Q(x)$ zusammen mit $\Sigma$ inkonsistent? $\Sigma \cup \{\neg P(x) \vee \neg Q(x)\}$
  - Inferenzverfahren um leere Klausel herzuleiten ist Resolution

# Hornlogik und Prolog

- Konvention Groß-/Kleinschreibung
  - Prädikatssymbole und Funktionssymbole aus Kleinbuchstaben
  - Variablensymbole aus Großbuchstaben

equals, add
mutter, f, g, p, q

X, Y, Z, X0, Y1
A, B, P, Q

- Fakten
  - als atomare Formel gefolgt von Punkt(.)

```
mag(hans, brot).
equals(X, X).
```

- Regeln als Implikation
  - Konklusion (*Head*) zuerst
  - Visualisierung der Implikation ⇐ durch :-
  - Prämissen (*Body*) danach kommasepariert

```
mag(hans, X) :- scharf(X).
equals(X, Y) :-
  equals(Y, X).
equals(X, Z) :-
  equals(X,Y), equals(Y,Z).
```

$\text{Equals}(x,y) \wedge \text{Equals}(y,z) \Rightarrow \text{Equals}(x,z)$

- Anfrage
  - Beginnt mit ?-
    Das Fragezeichen um An*frage* zu visualisiern
  - Atomare Formeln danach kommasepariert

```
?- mag(hans, chili).
?- equals(a, X), equals(X, b).
```

# Hornlogik und Prolog – Beispiele

Indian(Curry).
...

$\forall x\ \text{Indian}(x) \wedge \text{Mild}(x) \Rightarrow \text{Likes}(\text{Sam}, x)$

Gilt $\Sigma \models \text{Likes}(\text{Sam}, \text{Dahl})$ ?

```
indian(curry).
indian(dahl).
indian(tandoori).
indian(kurma).
mild(dahl).
mild(tandoori).
mild(kurma).
chinese(chow_mein).
chinese(chop_suey).
chinese(sweet_and_sour).
italian(pizza).
italian(spaghetti).
```

Fakten

$\Sigma$

```
likes(sam, Food) :-
        indian(Food), mild(Food).
likes(sam, Food) :- chinese(Food).
likes(sam, Food) :- italian(Food).
likes(sam, chips).
```

Regeln

```
?- likes(sam, dahl).
?- likes(sam, chop_suey).
?- likes(sam, pizza).
?- likes(sam, chips).
?- likes(sam, curry), indian(tandoori).
```

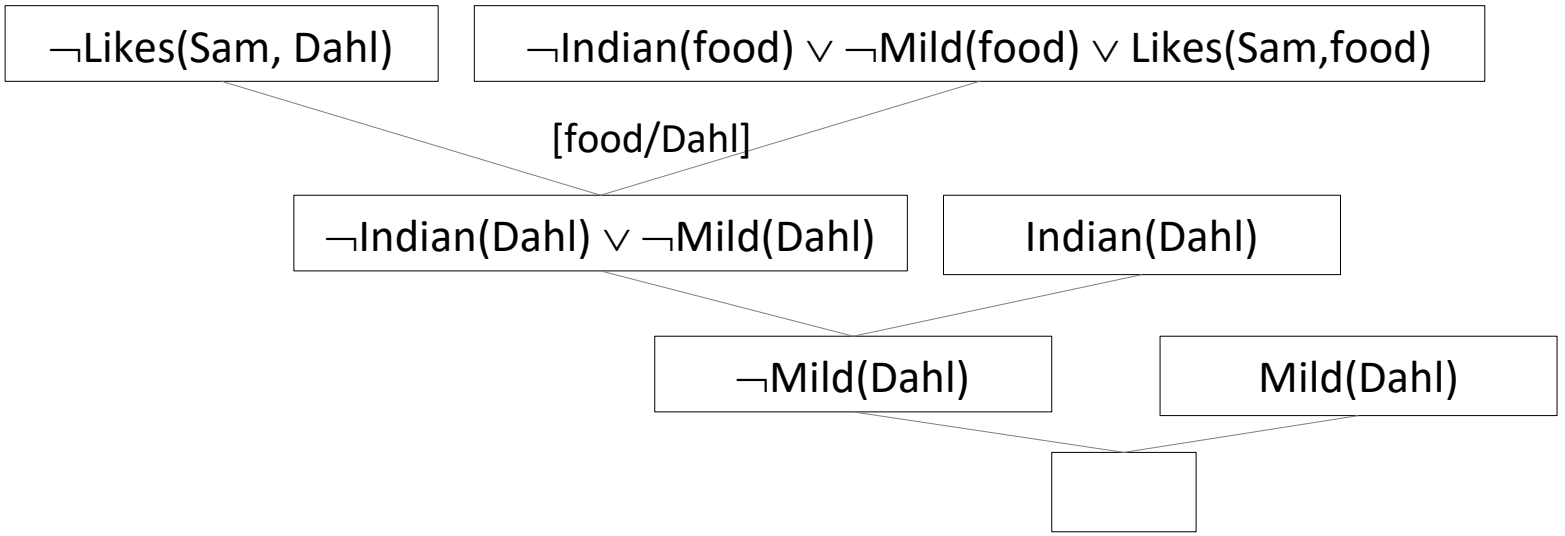
Anfragen

# Hornlogik – Inferenzverfahren

- Inferenzverfahren für $\Sigma \models \beta_1 \wedge \ldots \wedge \beta_n$
  - Ziel ist immer das Zeigen der Inkonsistenz von $\Sigma \cup \{\neg\beta_1 \vee \ldots \vee \neg\beta_n\}$
  - Dazu ist in der Hornlogik Resolution ausreichend (Faktorisierungsregel ist nicht notwendig)
  - Spezielle Strategie zur Regelauswahl
- SLD-Resolution
  - Selective Linear Resolution with Definite clauses
  - Suchbaum für Resolution von einer Anfrage
    - Auswahl der möglichen Regeln, immer aus Anfrage, Reihenfolge unbestimmt
  - Resolventen sind wieder Hornklauseln
- Umsetzung in Prolog
  - Literale der Anfrage von links nach rechts
  - Regelalternativen von oben nach unten, Tiefensuche
- Anfrage implizit existenzquantifiziert
  - Man sucht ein „Beispiel“ für die Inkonsistenz
  - Beispiel ist Antwortsubstitution, die Kumulation der Unifikatoren

# Hornlogik/Prolog – Beispiel

- Gilt $\Sigma \models$ Likes(Sam, Dahl)?




- ```
  ?- likes(sam, dahl).

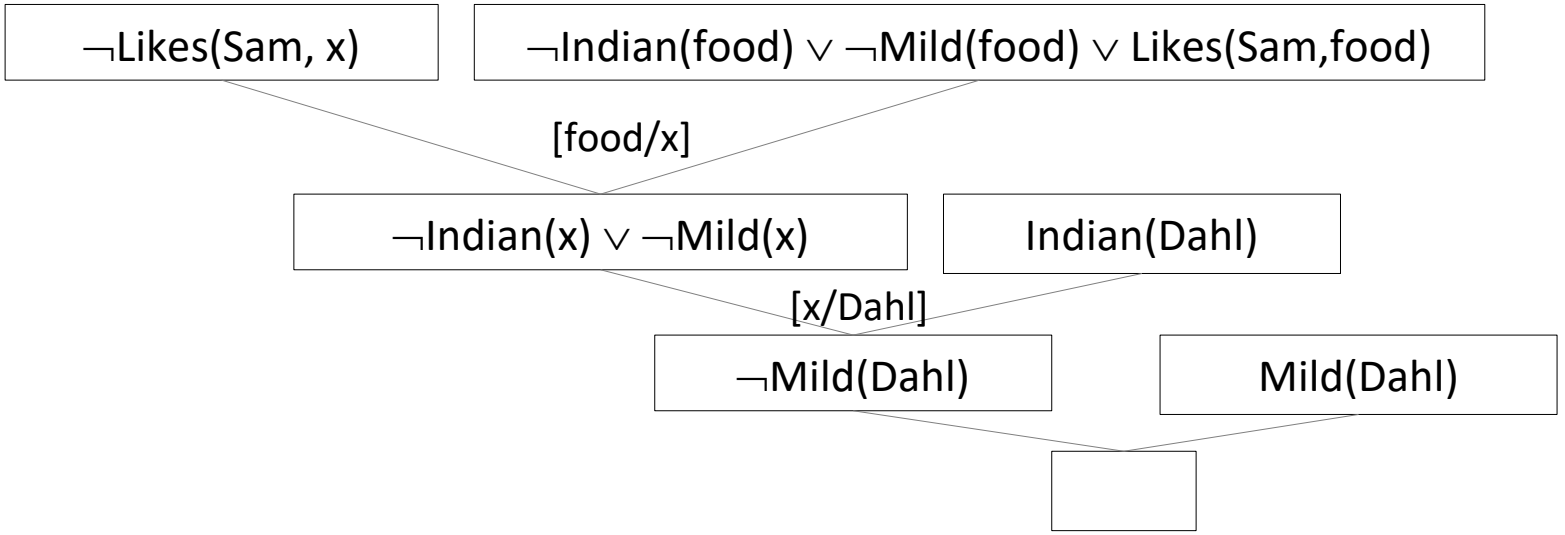
  ?- indial(dahl), mild(dahl).
  ?- mild(dahl).
  ?-

  yes
  ```

```
likes(sam, Food) :-
  indian(Food), mild(Food).
indian(dahl).
mild(dahl).
```
```

# Beispiel mit Antwortsubstitution

- Gilt $\Sigma \models$ Likes(Sam, x)?




- `?- likes(sam, X).`

```
?- indial(dahl), mild(dahl).
?- mild(dahl).
?-

X=dahl ?
```

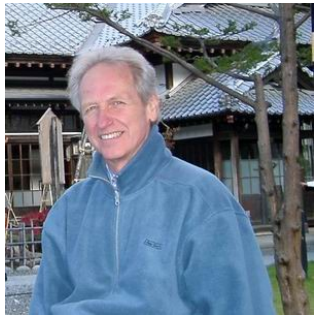
```
likes(sam, Food) :-
  indian(Food), mild(Food).
indian(dahl).
mild(dahl).
```

weitere mögliche Antwortsubstitutionen auf Rückfrage

# Logische Programmierung

- Deklarative Programmierung
  - Programmierparadigma (wie imperativ, funktional, objektorientiert)
  - Problembeschreibung statt Lösungsweg
  - Die Umgebung findet die Lösung
- Logische Programmierung
  - ist deklarative Programmierung
  - Prädikatenlogische Formeln für Beziehungen zwischen Objekten
  - Formeln sind Problembeschreibung
  - Inferenzmaschine berechnet Lösung
- Logische Programmierung mit Prolog
  - Einschränkungen (Horn-Logik, links/rechts Tiefensuche SLD-Resolution) und damit Inferenzmaschine effizient realisierbar
  - Ungetypte Terme als Datenstrukturen
  - + weitere Kompromisse und/oder Constraint-Systeme

# Prolog



- Historie
  - Alain Colmerauer, Robert Kowalski, 1972
    Universität von Marseille, ursprüngliches Ziel:
    Verständnis natürlicher Sprache, Französisch
  - Prädikatenlogik erster Stufe, Regelsysteme
    Einschränkung und vollständiges Suchverfahren (SLD-Resolution)
    Ausführung Backtracking
  - 80er Jahre: Freie und kommerzielle Prolog-Implementierungen,
    Neben LISP die Sprache für KI
  - 90er Jahre: Japan 5th Generation Computer Projekt, Compiler/WAM
    Constraint Logic Programming (CLP)
  - Heute: Spezialanwendungen in KI, Logik, Deduktive Datenbanken, Symbolische Berechnungen, Operations Research, ...
    Ausgereifte Implementierungen (SICStus, SWI, ...)
- SWI-Prolog, http://www.swi-prolog.org/
  - Seit 1986, WAM-basiert, schnell und stabil, Plattformunabhängig
  - Graphische Oberfläche, CLP (Q, FD, ...)

# Programmieren

- Programmieren
  - Fakten abgeben
  - Regeln angeben
- Datenbasis laden
  - Dateiname mit .pl am Ende
  - Einlesen mit ?- [dateiname]. ohne .pl am Ende
  - Alternative mit consult/comile
- Anfragen stellen
  - Aussage hinter ?-

verwandschaft.pl

```
frau(carol).    % Carol ist eine Frau
frau(eva).
frau(susi).
frau(lilith).
mann(abel).     % Abel ist ein Mann
mann(adam).

elter(adam, kain). % Adam ist ein Elternteil von Kain
elter(eva, kain).
elter(kain, susi).
elter(carol, susi).
elter(lilith, carol).

mutter(X, Y) :-     % X ist eine Mutter von Y
    elter(X, Y),    % wenn X Elternteil von Y ist und
    frau(X).        % X eine Frau ist

oma(X, Z) :-        % X ist eine Oma von Z
    mutter(X, Y),   % wenn X eine Mutter von Y ist und
    elter(Y, Z).    % Y ein Elternteil von Z ist
```

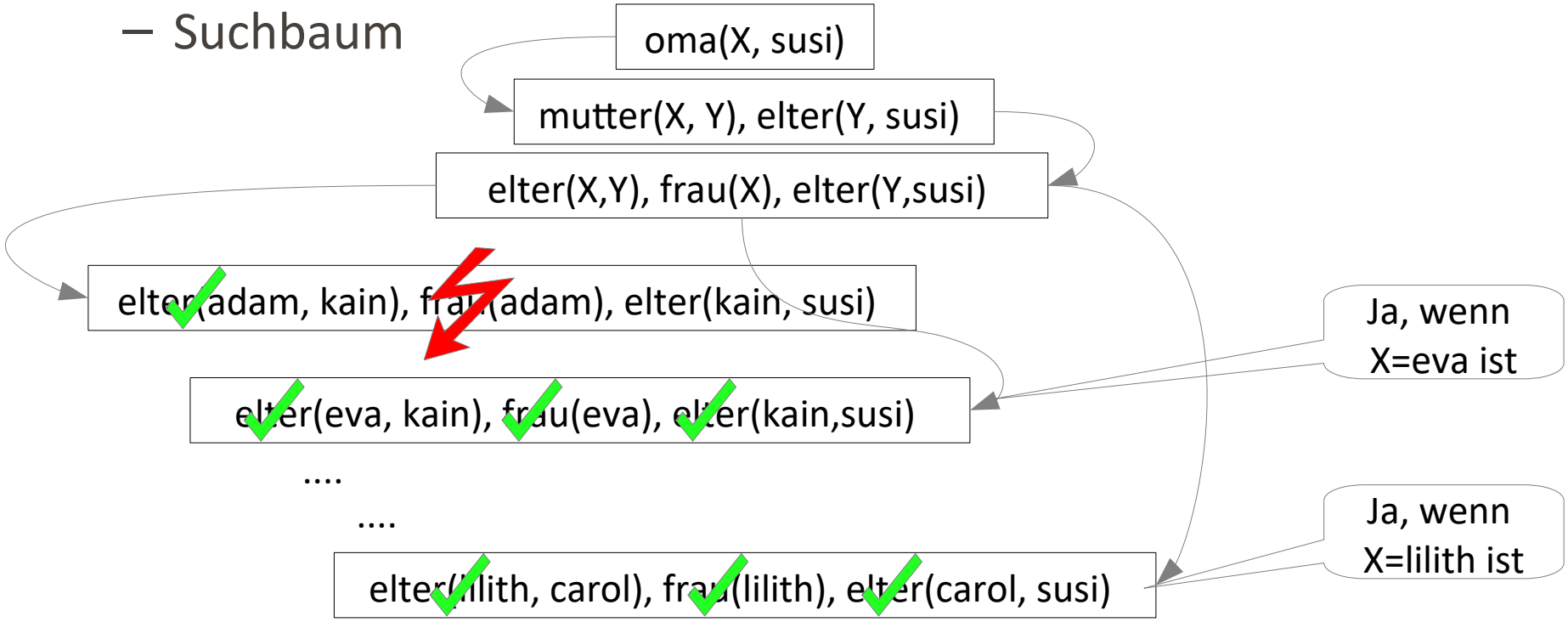
```
?- [verwandschaft].   % Laden
?- oma(X, susi).    % Welche X sind Oma von Susi?
X = eva ;           % Eva ist Oma von Susi
X = lilith ;        % Lilith ist Oma von Susi
No                  % sonst niemand
?-
```

# Anfrage und Berechnung

- Anfrage
  - Für welche X gilt X ist die oma von susi?

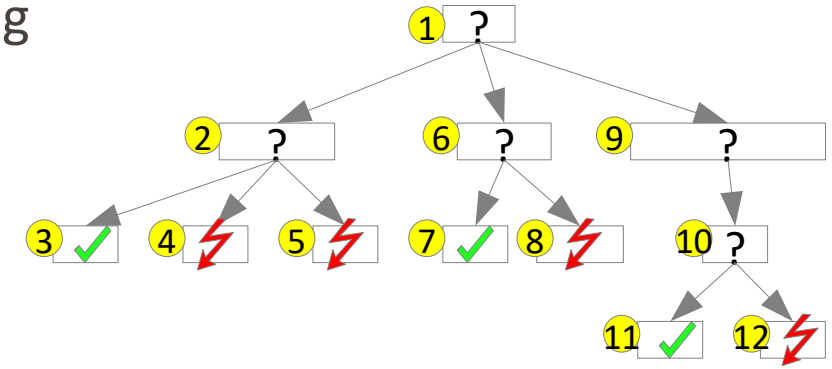
oma(X, susi)

- Berechnung
  - Probiere alle Regeln
  - Suchbaum



# Berechnung in Prolog

- Suche nach einer Variablenbelegung
  - Von links nach rechts
  - Tiefensuche, Backtracking
    - Kann schiefgehen, wenn ein Pfad unendlich lang wird




- Erfolg/Mißerfolg
  - Bei Erfolg „Ja“ und Ausgabe der aktuellen Variablenbelegung
    - Bei ; weiter probieren
    - Bei Return beenden
  - Bei Mißerfolg „Nein“

```
?- oma(X, susi).
X = eva ;
X = lilith ;
No
?-
```

```
% Welche X sind Oma von Susi?
% Eva ist Oma von Susi
% Lilith ist Oma von Susi
% sonst niemand
```

- Wenn nichts gefunden wurde „Nein“
  - Annahme, dass alles Wissen vorhanden ist
  - Closed World Assumption

```
?- elter(adam, carol).
No
```

Könnte sein, aber ist nicht in der Menge der Fakten.

# Mehrere Regeln – Vorfahren

- X ist Vorfahre von Z
  - Wenn X ein Elternteil von Z ist
  - Oder wenn X ein Elternteil von Y ist und Y ein Vorfahre von Z

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

- Beispiel:
  - kain, carol, ... lilith sind Vorfahren von susi
  - lilith ist der einzige Vorfahr von carol
  - eva ist Vorfahr von kain, abel und susi
- Achtung – Reihenfolge!
  - von links nach rechts mit Tiefensuche kann schief gehen

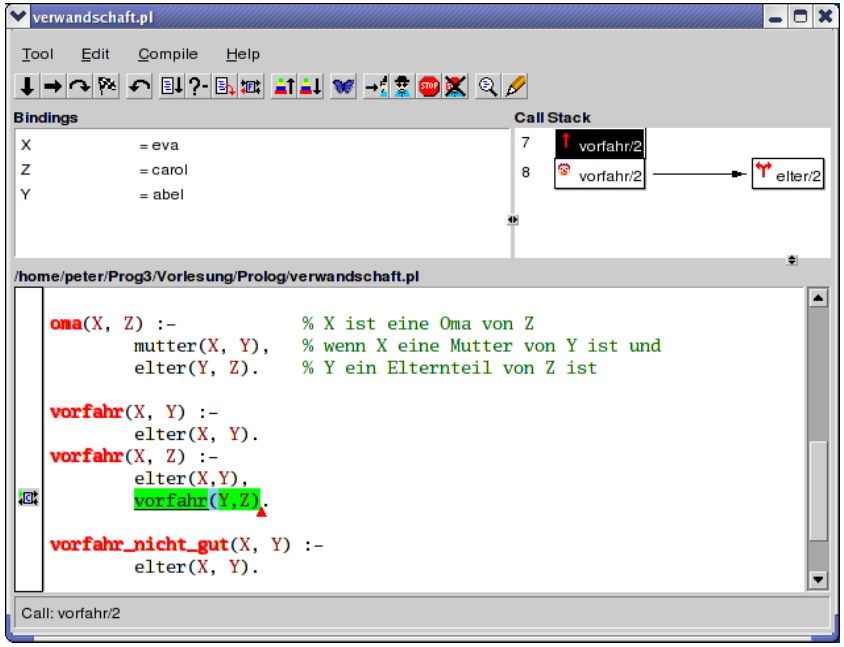
```
vorfahr_nicht_gut(X, Y) :-
      elter(X, Y).
vorfahr_nicht_gut(X, Z) :-
      vorfahr_nicht_gut(Y, Z),
      elter(X, Y).
```

```
?- vorfahr(X, susi).
X = kain ;
X = carol ;
X = adam ;
X = eva ;
X = lilith ;
No
?- vorfahr(X, carol).
X = lilith ;
No
?- vorfahr(eva, X).
X = kain ;
X = abel ;
X = susi ;
No
?-
```

```
?- vorfahr_nicht_gut(susi, X).
ERROR: Out of local stack
?- vorfahr_nicht_gut(X, abel).
X = adam ;
X = eva ;
ERROR: Out of local stack
?-
```

# Prolog bei der Ausführung beobachten

- Aufrufe und Variablen-belegung bei der Tiefensuche
  - Text-basiert mit trace.
  - Graphisch mit guitracer.

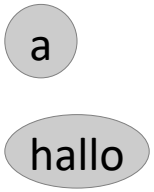



```
?- trace.
[trace]  ?- vorfahr(X, carol).
   Call: (8) vorfahr(_G315, carol) ? creep
   Call: (9) elter(_G315, carol) ? creep
   Exit: (9) elter(lilith, carol) ? creep
   Exit: (8) vorfahr(lilith, carol) ? creep
X = lilith ;
   Redo: (8) vorfahr(_G315, carol) ? creep
   Call: (9) elter(_G315, _L192) ? creep
   Exit: (9) elter(adam, kain) ? creep
   Call: (9) vorfahr(kain, carol) ? creep
   Call: (10) elter(kain, carol) ? creep
   Fail: (10) elter(kain, carol) ? creep
   Redo: (9) vorfahr(kain, carol) ? creep
   ...
   Redo: (10) vorfahr(susi, carol) ? creep
   Call: (11) elter(susi, _L214) ? creep
   Fail: (11) elter(susi, _L214) ? creep
No
?- guitracer.
% The graphical front-end will be used for subsequent tracing
Yes
?- trace.
Yes
[trace]  ?- vorfahr(X, carol).
```

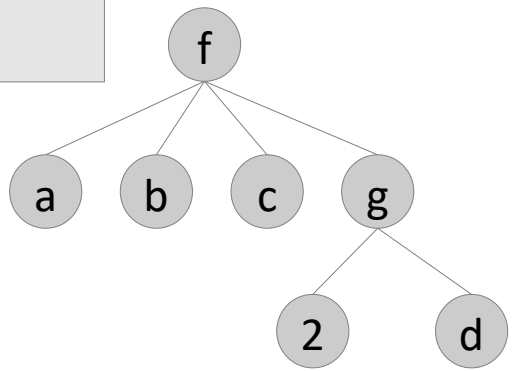
# Datenstrukturen sind Terme

- Atome
  - Kleingeschriebene Wörter
  - Funktoren ohne Parametern
- Terme
  - Baumstruktur
  - Funktoren mit mehreren Argumenten
  - Argumente sind Terme
  - Atome sind Terme
- Listen
  - Darstellbar als Baum
  - Spezielle Notation in Prolog möglich

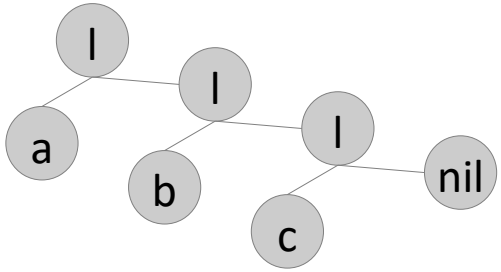
```
?- X = a, Y = hallo.
X = a
Y = hallo
```




```
?- X = f(a,b,c, g(2, d)).
X = f(a, b, c, g(2, d))
```




```
l(a, l(b, l(c, nil)))
```

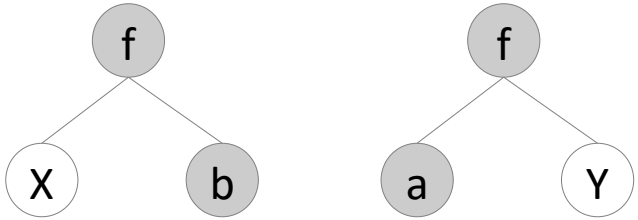


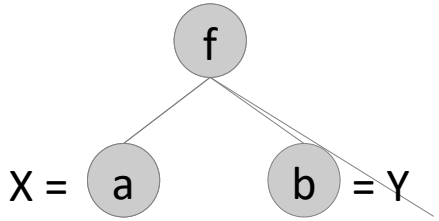

```
?- X = [a,b,c], Y = [a | [b | [c | []]]].
X = [a, b, c]
Y = [a, b, c]
Yes
```

# Unifikation

- =
  - Nicht Zuweisung sondern Unifikation
  - Versucht eine Variablenbelegung zu finden zwei Terme gleich zu machen
  - Variablen an beliebigen Stellen im Term
- Unifikationsmethode
  - Gleichungsmenge
  - Wenn Funktoren nicht gleich, FAIL
  - Ersetze Funktionsgleichung durch
  - Entferne triviale Gleichungen zwischen Atomen
  - Wenn eine Seite eine Variable ist, dann ersetze jedes Vorkommen der Variable durch andere Seite
  - Ergebnis wie Robinson-Unifkation
- Zyklische Terme vermeiden
  - Je nach Prolog erlaubt, mit Occurscheck verboten

```
?- f(X, a) = f(b, Y).
```




```
?- f(X) = X.
X = f(**)
Yes
?- unify_with_occurs_check(f(X),X).
No
```

# Unifikation – Beispiele

- Reihenfolge beliebig,
  Menge von Gleichungen,
  Vereinfachung
  - { X = f(Y), Y= a }
    { X = f(a), Y=a }
- Gleichsetzen der Unterterme,
  Vereinfachung
  - { f(X,a) = f(b,Y) }
    { X = b, a = Y }
  - { f(X, a, g(Y, X)) = f(c, a, Z) }
    { X = c, a = a, g(Y,X) = Z }
    { X = c, g(Y, X) = Z}
    { X = c, g(Y, c) = Z}
- Nicht immer eine Lösung
  - { f(X,b) = f(a,X) }
    { X =a, b = X}  % Widerspruch

```
?- X = f(Y), Y = a.
X = f(a)
Y = a
Yes
?- f(Y) = X, Y = a.
Y = a
X = f(a)
Yes

?- f(X, a) = f(b, Y).
X = b
Y = a
Yes

?- f(X,a,g(Y, X)) = f(c,a,Z).
X = c
Z = g(Y, c)
Yes

?- f(X, b) = f(a, X).
No
```

# Terminierung

- Terminierung nicht garantiert
  - Aufgrund der Strategie endlose Inferenzketten möglich
  - Im Beispiel liefert Anfrage
    `?- p(a).`
    kein Ergebnis
  - Es wird immer wieder (nach Umbenennung der Variablen X) mit der ersten Regel resolviert, aber kein Fortschritt erzielt

```
p(X) :- p(X).
p(a).
```

```
p(X) :- p(f(X)).
p(a).
```

```
?- p(a).
ERROR: Out of local stack
```

```
p(f(X)) :- p(X).
p(a).
```

wäre ok

- Wie beim Programmieren denken
  - Reihenfolge beachten
  - Problemreduktion,wie Rekursion
    - Problem muss kleiner werden Ordnung
    - Es darf nur endlich viele Schritte bis zum trivialen Fall geben, diskrete Ordnung

```
p(a).
p(X) :- p(X).
```

```
?- p(b).
ERROR:
```

```
p(a).
```

```
?- p(b).
false
```

nicht herleitbar

# Negation und Closed World Assumption

- Negation
  - Nicht unterstützt in Hornlogik
  - Nicht unterstützt in Prolog
- Negation as Failure
  - Wenn mit den vorhandenen Regeln eine Anfrage nicht bewiesen werden kann, dann wird angenommen die Anfrage gilt nicht
  - Statt Negation besser „nicht herleitbar“ $\nvdash P(x)$
  - Eingebautes Prädikat \+ sieht ähnlich aus wie $\nvdash$ (hieß früher not())
    - Metaprädikat (hat Prädikat als Argument)
    - Vermeiden!
  - ACHTUNG: Eine solche Aussage kann sich ändern, wenn neue Fakten hinzu kommen, also wird **nicht bewiesen** $\vdash \neg P(x)$ **keine Negation** in Hornlogik
- Closed World Assumption
  - Alles was nicht als wahr gezeigt werden kann ist falsch

```
p(a).
```

```
?- p(b).
false.
?- \+(p(b)).
true.
```

# Arithmetik

- Arithmetik
  - Repräsentation von natürlichen Zahlen durch Terme
    - 0 durch `0`
    - 1 durch `succ(0)`
    - 2 durch `succ(succ(0))`
    - …
  - Addition
    - Neutrales Element 0
    - x+y = (x-1) + (y+1)
    - (oft) terminierend, da kleiner werdend und gegen 0
    - Idealerweise „Sorte“ festlegen
- Anfragen
  - Rechnen
  - Aber auch umstellen!
  - Und Lösungen aufzählen!

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

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

```
add(0, X, X) :-
        zahl(X).
```

```
?- zahl(X).
X = 0 ;
X = succ(0) ;
X = succ(succ(0)) ;
X = succ(succ(succ(0))) .

?- add(succ(succ(0)), succ(0), X).
X = succ(succ(succ(0))).

?- add(succ(succ(0)), X, succ(succ(succ(succ(0))))).
X = succ(succ(0)).

?- add(X, Y, succ(succ(0))).
X = 0,   Y = succ(succ(0)) ;
X = succ(0),   Y = succ(0) ;
X = succ(succ(0)),   Y = 0 ;
false.
```

# Programmieren mit Listen – member

- member
  - X ist Element einer Liste, wenn es das erste Element der Liste ist
  - X ist Element einer Liste, wenn es in der Liste außer dem ersten ist
- Kontrollstrukturen
  - Rekursion statt Schleifen
  - Mehrere Regeln statt Verzweigung
- Verwendung
  - Test
  - Aufzählung
- Achtung
  - Kann durch starre Tiefensuche unendlich lange laufen

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

```
?- member(a,[a,b,c]).
Yes
?- member(c,[a,b,c]).
Yes
?- member(d,[a,b,c]).
No
?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
No
?- member(a,L).
L = [a|_G246] ;
L = [_G245, a|_G249] ;
L = [_G245, _G248, a|_G252] ;
L = [_G245, _G248, _G251, a|_G255];
...
```

# Programmieren mit Listen – append

- append – Listen zusammenfügen
  - Eine Liste an die leere Liste angefügt ist die Liste
  - Das erste Element der ersten Liste ist das erste Element der zusammengefügten Liste
  - Die Restliste der zusammengefügten Liste ist die erste Liste ohne erstes Element angefügt an die zweite Liste

```
append([], L, L).
append([X|L0], L1, [X|L2]) :-
   append(L0, L1, L2).

% append ist eingebaut in SWI-Prolog
```

- Beispiel ?- append([a,b], [c,d], L).

?- *X=a, L0=[b], L1=[c,d], L=[a|L2],*
append([b], [c,d], L2).

Neue Variablen in Regel 2 und Anwendung Regel 2

?- X =a, L0=[b], L1=[c,d], L=[a,*b|L2'*],
*X' =b, L0'=[], L1'=[c,d], L2=[b|L2'],*
append([], [c,d], L2').

Neue Variablen in Regel 2 und Anwendung Regel 2

?- X =a, L0=[b], L1 =[c,d], L= [a,b,*c,d]*,
X' =b, L0'=[], L1'=[c,d], L2= [b,*c,d],*
L2'=[c,d].

Neue Variablen in Regel 1 und Anwendung Regel 1, keine weiteren Prädikate

?- L = [a,b,*c,d]*.

Lösung nach Elimination nicht sichtbarer Variablen

# Beispiele mit append

- Füge zwei Listen zusammen
- Welche Liste muss man anfügen?
- Welche Liste, außer dem ersten Element, muss man anfügen?
- Welches ...
- Welche Möglichkeiten gibt es zwei Listen zusammenzufügen um eine vorgegebene Liste zu erhalten?

```
?- append([a,b,c], [d,e,f], Z).
Z = [a, b, c, d, e, f]

?- append([a,b,c], X, [a,b,c,d,e,f]).
X = [d, e, f]

?- append([a,b,c], [d|X], [a,b,c,d,e,f]).
X = [e, f]

?- append([a,b,c], [X|[e,f]], [a,b,c,d|Z]).
X = d
Z = [e, f]

?- append(X, Y, [a,b,c,d,e,f]).
X = []              Y = [a, b, c, d, e, f] ;
X = [a]             Y = [b, c, d, e, f] ;
X = [a, b]          Y = [c, d, e, f] ;
X = [a, b, c]       Y = [d, e, f] ;
X = [a, b, c, d]    Y = [e, f] ;
X = [a, b, c, d, e] Y = [f] ;
X = [a, b, c, d, e, f] Y = [] ;
```

# Weitere Listen-Prädikate

- Listen-Bibliothek
  - Wird in SWI-Prolog automatisch bei Bedarf geladen
  - Viele sinnvolle Listen-Prädikate
  - Parameter in Doku. annotiert: ? Ein/Ausgabe, + Eingabe, - Ausgabe
    Hinweis auf sinnvolle Verwendung
- Auszug
  - `nth0(?Index, ?List, ?Elem)`
    Elem ist Element der Liste an Stelle Index (ab 0 gezählt)
    nth1 wie nth0 nur ab 1 gezählt
  - `delete(+List1, ?Elem, ?List2)`
    In List2 sind alle Elemente von List1 außer Elem,
    List1 muss instanziierte Liste sein
  - `select(?Elem, ?List, ?Rest)`
    Elem ist Element der Liste, Rest ist Liste ohne Elem
  - `permutation(?List1, ?List2)`
    List1 ist eine Permutation von List2

# Cut

- Ausführungsstrategie
  - Anfrage von links nach rechts
  - Regeln von oben nach unten
  - Tiefensuche
- Problem
  - Endloser Abstieg bei Tiefensuche
  - Absehbare erfolglose Suche in Teilbaum
- Lösung
  - Tiefensuche abschneiden
  - Cut-Operator, !
- Cut
  - Achtung, keinerlei logische Entsprechung ausschließlich operational
  - Man verliert Möglichkeit aufzuzählen
  - Vermeiden

```
?- lebt(X).
X = rose;
X = lilie;
X = hund;
...
```

```
lebt(X) :- blume(X).
lebt(X) :- tier(X).
blume(rose).
blume(lilie).
tier(hund).
tier(katze).
tier(maus).
```

```
[trace]  ?- lebt(lilie).
   Call: (6) lebt(lilie) ? creep
   Call: (7) blume(lilie) ? creep
   Exit: (7) blume(lilie) ? creep
   Exit: (6) lebt(lilie) ? creep
true ;
   Redo: (6) lebt(lilie) ? creep
   Call: (7) tier(lilie) ? creep
   Fail: (7) tier(lilie) ? creep
   Fail: (6) lebt(lilie) ? creep
false.
```

Wenn was eine Blume ist, dann ist es kein Tier, absehbar erfolglos

```
lebt(X) :- blume(X),!.
```

```
[trace]  ?-
```

```
?- lebt(X).
X = rose.
?-
```

```
[trace]  ?- lebt(lilie).
   Call: (6) lebt(lilie) ? creep
   Call: (7) blume(lilie) ? creep
   Exit: (7) blume(lilie) ? creep
   Exit: (6) lebt(lilie) ? creep
true.
```

# Cut – Beispiel

- Beispielprogramm
  - Einstellige Prädikate p, q, r
- Beispielanfragen
  - p(1).
    - Ja
    - Cut wird nicht abgearbeitet
    - p(1) als Fakt führt zum Erfolg
  - p(2).
    - Ja
    - Cut wird abgearbeitet
    - Erfolg wegen r(2)
  - p(3).
    - Nein
    - Cut wird abgearbeitet
    - Cut verhindert Backtracking

```
p(X) :-
        q(X),
        !,
        r(X).
p(1).
p(2).
p(3).
q(2).
q(3).
r(2).
```

# Cut und Negation

- Negation as Failure
  - Wir nehmen an, dass not(P(X)) gilt, wenn P(X) nicht beweisbar
- Selbst implementierbar
  - cut verwenden
  - call verwenden
    - Ruft ein Prädikat, versucht eine Aussage zu beweisen
    - Metaprädikat, das Eval von Prolog
  - Versucht P zu zeigen
    - Wenn es klappt, dann Cut (nicht mehr über Stelle zurück Backtracking) fail, forciert Fehlschlag
    - Wenn es nicht klappt, dann zweite Klausel; es klappt
- Nicht sehr intuitiv
  - Zählt zum Beispiel nicht auf
  - Vermeiden, nicht selbst machen sondern \+ nehmen

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

```
?- not(lebt(lilie)).
false.

?- not(lebt(haus)).
true.

?- not(lebt(X)).
false.
```

# Meta-Prädikate

- Meta-Prädikate
  - Prädikate
  - Arbeiten mit Prädikaten als Argumenten statt Termen
  - Nicht mehr Prädikatenlogik erster Stufe
  - Nur für Spezialaufgaben, vermeiden
  - Bekannte Beispiele: call, not
- Weitere Beispiele
  - apply(:Goal, +List): Fügt Listenelemente als Parameter an Goal an und ruft es
  - call_with_depth_limit(:Goal, +Limit, -Result): Tiefenbeschränkte Suche, für iterative deepening
  - findall(+Template, :Goal, -Bag): Sucht alle Lösungen für Goal und sammelt in Bag die Bindungen von Template für jede Lösung
  - … spezifisch je Implementierung, Dokumentation

In Dokumentation Parameterannotation:
+ Eingabe, instanziiert
- Ausgabe, Variable
? Ein/Ausgabe
: Prädikat

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

```
?- findall(X, append(X, Y, [1,2,3,4]), L).
L = [[], [1], [1, 2],
    [1, 2, 3], [1, 2, 3, 4]].
```

# Eingebaute Arithmetik

- Berechnung
  - = ist Termgleichheit/Unifikation
  - is für Auswertung und Zuweisung
- Arithmetische Vergleichsoperationen
  - <,>,>=
    wie gewohnt
  - =< statt <=!
    (<= als Implikation verwendet)
  - Arithmetische Gleichheit
    =:= gleich
    =\= ungleich
- Nur Grundterme
  - Achtung: Keine Variablen in arithmetischen Ausdrücken

```
? 2+3 = 3+2.
No
?- X is 3+2.
X = 5

?- 2+3 >= 1+5.
No
?- 2+5 >= 1+5.
Yes
?- 2+3 < 1+5.
Yes
?- 2+3 >= 3+1.
Yes

?- 3+2 =:= 2+3.
Yes

?- X = 3, X+2 =:= 2+X.
X = 3
Yes
?- X+2 =:= 2+X, X = 3.
ERROR: =:=/2: Arguments are not
sufficiently instantiated
```

```
? X = 3+2.
X = 3+2
?- X is 3+2, X=5.
X = 5

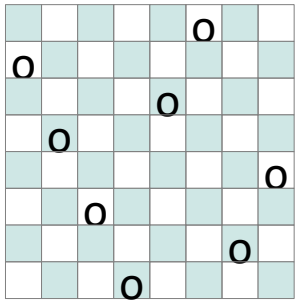
?- 2+3 > 3+1.
Yes
?- 2+3 <= 3+4.
ERROR: Syntax error:
Operator expected

?- 2+3 =< 3+4.
Yes

?- 3+2 =\=
2+3.
No
?- 3+2 =\=
2+4.
Yes
```

# Beispiel – N-Damen

- N-Damen Problem
  - N Damen auf einem Schachbrett der Größe NxN verteilen (Alg. u. Datenstrukturen)
  - Generieren und Testen



```
?- ndamen(8,Damen).
Damen = [4, 7, 3, 8, 2, 5, 1, 6]
Yes

?- findall(Damen, ndamen(8,Damen), Loes),
   length(Loes, LenLoes).
Loes = [[4, 7, 3, 8, 2, 5, 1, 6], ...]
LenLoes = 92
```

```
ndamen(N, Ds) :-
   range(N, NL, Ds),
   permutation(NL, Ds), % generate
   sicher(Ds).        % test

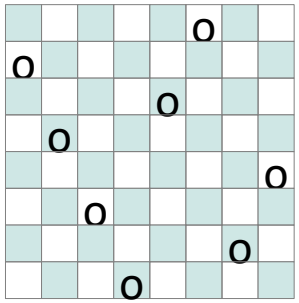
% Zahlen 1..N, N Variablen
range(0, [], []).
range(N, [N|L], [_|Ds]) :-
   N >= 0,
   N1 is N-1,
   range(N1, L, Ds).

sicher([]).
sicher([D|Ds]) :-
   sicher(Ds, 1, D),
   sicher(Ds).

% Dame sicher wenn Diagonale frei
sicher([], _, _).
sicher([TD|Ds], N, D) :-
   TD + N =\= D,
   TD - N =\= D,
   N1 is N+1,
   sicher(Ds, N1, D).
```

# Beispiel – N-Damen

- N-Damen Problem
  - N Damen auf einem Schachbrett der Größe NxN verteilen (Alg. u. Datenstrukturen)
  - Generieren und Testen



```
?- ndamen(8,Damen).
Damen = [4, 7, 3, 8, 2, 5, 1, 6]
Yes

?- findall(Damen, ndamen(8,Damen), Loes),
   length(Loes, LenLoes).
Loes = [[4, 7, 3, 8, 2, 5, 1, 6], ...]
LenLoes = 92
```

```
ndamen(N, Ds) :-
   range(N, NL, Ds),
   permutation(NL, Ds), % generate
   sicher(Ds).        % test

% Zahlen 1..N, N Variablen
range(0, [], []).
range(N, [N|L], [_|Ds]) :-
   N >= 0,
   N1 is N-1,
   range(N1, L, Ds).

sicher([]).
sicher([D|Ds]) :-
   sicher(Ds, 1, D),
   sicher(Ds).

% Dame sicher wenn Diagonale frei
sicher([], _, _).
sicher([TD|Ds], N, D) :-
   TD + N =\= D,
   TD - N =\= D,
   N1 is N+1,
   sicher(Ds, N1, D).
```