Überblick 2. Vorlesungsteil

- Deduktives Schließen
  - Deduktive Datenbanken, Datalog
  - Kurze Einführung in Prolog
    - Beweisbäume
- Induktives Schließen
  - Explanation-Based Learning
    - Automatisches Lernen von Datalog-Programmen durch Generalisierung von Beweis-Bäumen
  - Einführung in die induktive Logische Programmierung
    - Automatisches Lernen von Datalog-Programmen aus Trainings-Beispielen
- Data und Web Mining
  - Kurze Begriffsklärung
- Semantic Web
  - Einführung in XML(-Schema), RDF(-Schema), OWL

Deduktives Schließen

- Einführung in Datalog und Prolog
  - Fakten und Queries
  - Konjunktionen
  - Regeln, Theorien, Programme
- Semantik von Datalog
  - Beweisführung in Datalog (EPP)
  - Fixpunktsemantik
  - Datalog und Relationale Algebra
- Prolog
  - Erweiterungen von Prolog (Cut, Listen, Funktionen)
  - Deklarative Semantik vs. Prozedurale Semantik
  - SLD-Resolution
  - Beweisbäume
- Meta-Interpreter in Prolog

Prolog = Programming + Logic

- Prolog ist eine mächtige Programmiersprache
  - in der künstlichen Intelligenz recht beliebt
  - hauptsächlich in Europa und Japanvom japanischen 5th Generation Project gepusht

Datalog = Databases + Logic

- Datalog ist eine einfache Version von Prolog
  - etliche komplexe Features von Prolog wurden weggelassen
  - Fokus auf Ergänzung von Datenbanken durch deduktive Fähigkeiten

Historische Entwicklung

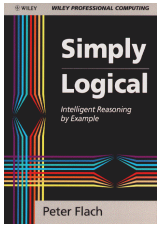
- 1965: Automatisches Beweisen durch Resolution (J. A. Robinson)
- Anfang 70-er: Erste Ansätze für Prolog (Kowalski, Colmerauer)
- 1977: First Workshop on Logic in Databases
- 1983: Warren Abstract Machine (WAM) für Prolog Pogramme
- 80-er: Japanese Fifth Generation Project
- 90-er: Constraint Logic Programming extensions.

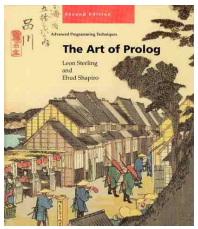
Prolog und Datalog-Systeme

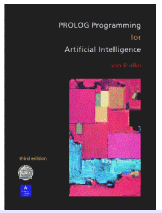
- Datalog Educational System (DES)
  - Einfache Implementierung von Datalog
  - basiert auf Prolog
  - erhältlich für die gängigsten Prolog-Systeme
  - auch als Windows-Executable
    - NEU: Java-Implementation mit GUI (acide)
  - http://des.sourceforge.net/
- SWI-Prolog
  - Freie GPL Prolog Implementation
  - http://www.swi-prolog.org/
- Gnu Prolog
  - natürlich ebenfalls frei
  - http://www.gnu.org/software/gprolog/gprolog.html

Literatur

- Artikel
  - S. Ceri, G. Gottlob, L. Tanca: What you always wanted to Know About Datalog (And Never Dared to Ask), *IEEE Transactions on Knowledge and Data Engineering* 1(1):146-166, 1989.
  - J. Grant, J. Minker: The Impact of Logic Programming on Databases. *Communications of the ACM* 35(3):66-81, 1992.
  - F. S. Perez: Datalog Educational System, User's Manual, 2004 - 2009.
- Bücher
  - H. Gallaire and J. Minker (eds.): *Logic and Databases,* Plenum 1978. (Proceedings of 1977 Workshop)
  - P. Flach: *Simply Logical - Intelligent Reasoning by Example*, John Wiley 1994. (gutes Lehrbuch für Deduktion und Induktion in Logik) PDF-download unter http://www.cs.bris.ac.uk/~flach/SimplyLogical.html
  - W. F. Clocksin and C. S. Mellish: *Programming in Prolog.* Springer-Verlag 1981. (Klassiker 1)
  - L. Sterling, E. Shapiro: The Art of Prolog, MIT Press, 2nd ed., 1994. (Klassiker 2)
  - I. Bratko: PROLOG Programming for Artificial Intelligence. Prentice Hall, 3rd ed., 2000. (Fokus auf künstliche Intelligenz) http://cwx.prenhall.com/bookbind/pubbooks/bratko3_ema/





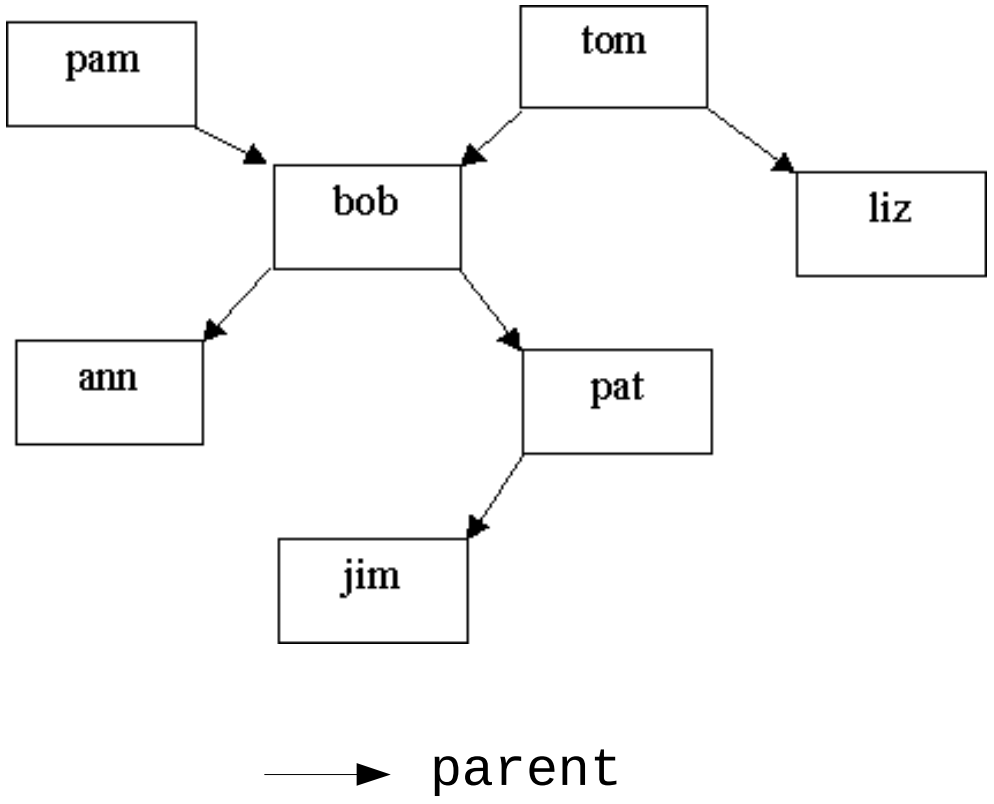


Beispiel – Family Relations

- Datenbank Relation parent

Parent

Elternteil	Kind
Pam	Bob
Tom	Bob
Tom	Liz
Bob	Ann
Bob	Pat
Pat	Jim



- Andere Schreibweise

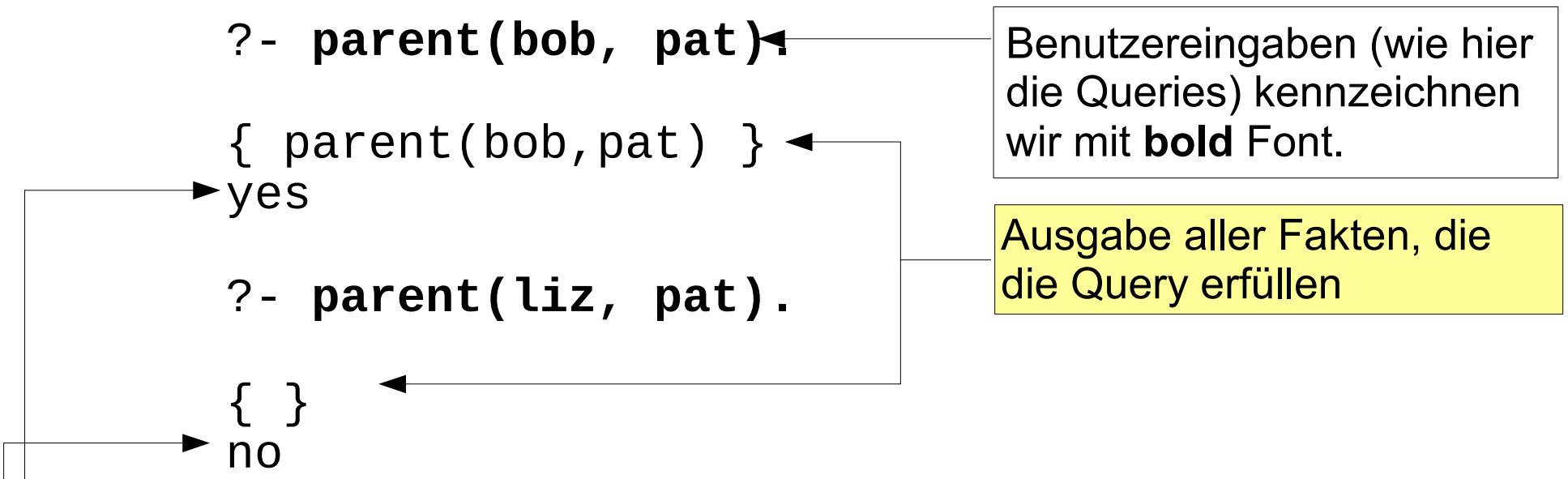
```
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
```

Definitionen

- **Konstante:** Ein Symbol, das ein Objekt repräsentiert.
  - beginnt mit einer Zahl oder einem Kleinbuchstaben.
    - z.B. `pam, bob, liz, 1, pi, true,` etc.
- **Prädikat:** Ein Symbol, das eine Relation zwischen Objekten beschreibt.
  - beginnt mit einem Kleinbuchstaben
    - z.B. `parent, male, female`
  - Die *Arität* eines Prädikats gibt die Anzahl der Stellen der Relation wieder
    - wird oft an den Namen des Prädikats angehängt
    - z.B. `parent/2, male/1, female/1`
- **Fakt:** Ein Fakt beschreibt einen Sachverhalt
  - Fakten bestehen aus
    - dem Prädikat
    - in Klammern einer Anzahl von Argumenten (entsprechend der Arität)
    - abgeschlossen mit einem Punkt
  - z.B. `parent(pam,bob).`

Einfache Queries

- An die Fakten können einfache Anfragen gestellt werden:



- Queries werden mit logischen Schlüssen bewiesen
  - Antwort `yes`: Faktum kann bewiesen werden bzw. findet sich in der Datenbank
  - Antwort `no`: Faktum kann nicht bewiesen werden bzw. findet sich nicht in Datenbank
- Queries werden auch als *Goals* bezeichnet.

Variablen

- **Variable:** Ein Symbol, das für eine nicht spezifizierte Konstante steht
  - beginnt mit einem Großbuchstaben oder einem Underscore
  - z.B. `X, Person, Nummer, _42,` etc.

- Gleiche Variablen-Symbole bezeichnen das gleiche Objekt!
  - Spezialfall: in Prolog bezeichnet die **anonyme Variable** "_" jedes Mal ein anderes Objekt.

- Semantik in Queries:
  - z.B. `?- parent(X,liz).`
  - **Bedeutung:**
    Welche `X` stehen mit `liz` in der Relation `parent`?

Beispiele in Datalog

- Wer ist ein Elternteil von Liz?

```
?- parent(X,liz).

{
  parent(tom, liz)
}

yes
```

- Für wen ist Bob ein Elternteil?

```
?- parent(bob,X).

{
  parent(bob, ann),
  parent(bob, pat)
}

yes
```

Beispiele in Prolog

- Wer ist ein Elternteil von Liz?

```
?- parent(X,liz).
X = tom
```

Ausgabe der (ersten) Variablenbelegung, die die Query erfüllt

- Für wen ist Bob ein Elternteil?

```
?- parent(bob,X).
X = ann ;
X = pat ;
No
```

Eingabe eines Strichpunkts frägt nach zusätzlichen Lösungen an

Keine weiteren Lösungen

Beispiele in Datalog

- Wer steht zu wem in Parent Relation?

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

{
  parent(bob, ann),
  parent(bob, pat),
  parent(pam, bob),
  parent(pat, jim),
  parent(tom, bob),
  parent(tom, liz)
}

yes
```

Ausgabe aller Fakten, die die Query erfüllen.

Weitere Definitionen

- **Term:**
  - eine Konstante oder eine Variable
  - *Anmerkung:* In Prolog gibt es auch noch Funktionssymbole
- **Atom:**
  - besteht aus einem Prädikatensymbol
  - und *n* Termen
  - Beispiel:

```
parent(X,liz).
```

- **Literal:**
  - besteht aus einem Atom oder der Negation eines Atoms
  - Schreibweisen:
    - `not(`*atom*`)`
    - `\+` *atom*
  - auf die genaue Semantik der Negation werden wir noch eingehen

Beispiele in Prolog

- Wer steht zu wem in Parent Relation?

```
?- parent(X,Y).
X = pam
Y = bob ;
X = tom
Y = bob
```

Ausgabe einzelner Variablenbindungen, die die Queries erfüllen. In der Reihenfolge, in der die Fakten in der Datenbank stehen.

- Welche Kinder finden sich in der Datenbank?

```
?- parent(_,Y).
Y = bob ;
Y = bob ;
Y = liz
Yes
```

Ergebnisse der anonymen Variable _ werden nicht gelistet

Aber man erhält dennoch ein Ergebnis für jedes Fakt

Eingabe von Return zeigt an, daß keine weitere Lösung gewünscht wird.

Konjunktionen

- Zwei oder mehrere Atome können mit einem logischen UND verknüpft werden
  - *Schreibweise*: Verbinden der Atome mit einem Komma
- gleiche Variablen bezeichnen dabei gleiche Objekte
- Beispiel: `parent(X,Y), parent(Y, pat).`
-
  - Bedeutung:
    - `X` ist ein Elternteil von `Y`, und `Y` ist ein Elternteil von `pat`.

    → `X` ist ein Großelternteil von `pat`.

Konjunktion und Join

- Eine Konjunktion entspricht im Prinzip einem (Natural) Join zweier Datentabellen
  - in unserem Beispiel einem Join der Tabelle mit sich selbst
- Anderes Beispiel:
  - `parent(X,Y), male(X).`

Male

Person
Tom
Bob
Jim

Female

Person
Pam
Pat
Liz
Ann

Parent

Elternteil	Kind
Pam	Bob
Tom	Bob
Tom	Liz
Bob	Ann
Bob	Pat
Pat	Jim

Parent ⋈ Male

Elternteil = Person

Elternteil	Kind
Tom	Bob
Tom	Liz
Bob	Ann
Bob	Pat

Beispiele in Prolog

- Wer sind die Großeltern von Pat?

```
?-  parent(X,Y), parent(Y,pat).
X = pam
Y = bob ;
X = tom
Y = bob
```

Die Eltern von Pat werden an die Variable Y gebunden, und ihre Eltern dann an X. Y muß daher benannt werden und wird auch ausgegeben.

- Welche Geschwister finden sich in der Datenbank?

```
?- parent(X,Y), parent(X,Z), Y \= Z.
X = tom
Y = bob
Z = liz ;
X = tom
Y = liz
Z = bob ;
X = bob
Y = ann
Z = pat
```

\= steht für ungleich!

Wiederum gibt es ein Ergebnis für jeden möglichen Beweis (jede mögliche Faktenbelegung)

Beispiele in Datalog

nächste Folie!

- Wer sind die Großeltern von Pat?

```
?-  parent(X,Y), parent(Y,pat).
    Info: Processing:
        answer(X,Y) :- parent(X,Y),parent(Y,pat)
 {
     answer(pam,bob),
     answer(tom,bob)
 }
```

Definiert eine Regel für ein neues virtuelles Prädikat answer/2 mit den Variablen X und Y.

Gibt dafür die Fakten aus, die die Definition erfüllen-

- Welche Geschwister finden sich in der Datenbank?

```
?- parent(X,Y), parent(X,Z), Y \= Z.
   Info: Processing:
       answer(X,Y,Z) :- parent(X,Y),parent(Y,Z),Y\=Z.
{
  {
      answer(bob,ann,pat),
      answer(bob,pat,ann),
      answer(tom,bob,liz),
      answer(tom,liz,bob)
   }
}
```

Wiederum gibt es ein Ergebnis für jeden möglichen Beweis (jede mögliche Faktenbelegung)

Regeln

- Man kann für die Ergebnis-Menge auch einen neuen Namen vergeben
  - und dadurch eine (virtuelle) Relation definieren

- Formal stellt eine Regel eine logische Implikation dar:
  - $A \wedge B \rightarrow C$
  - Wenn A und B gelten, dann gilt auch C.

B	H	B→ H
0	0	1
0	1	1
1	0	0
1	1	1

$$B \rightarrow H = \neg B \vee H$$

- Schreibweise:
  - In Prolog schreibt man traditionell die Implikation “verkehrt” herum:
    - `C :- A, B.`

Head der Regel:
Alles links der Implikation

Body der Regel:
Alles rechts der Implikation

Beispiele

- X ist der Vater von Y:
  - `father(X,Y) :- parent(X,Y), male(X).`
    - Lies:
      - X ist der Vater von Y, wenn X ein Elternteil von Y ist und X männlich ist.
    - `father(X,Y)` listet alle Väter mit ihren Kindern
- Nicht alle Variablen, die im Body vorkommen, müssen auch im Head vorkommen
  - `father(X) :- parent(X,Y), male(X).`
    - `father(X)` listet alle Väter
- Dieselbe Relation kann in der Definition auch mehrmals vorkommen
  - `grandparent(X,Y) :- parent(X,Z), parent(Z,Y).`
    - `grandparent(X,Y)` listet alle Großeltern mit ihren Enkeln

Verwendung von Regeln

- Neu definierte Relationen können genauso wie Datenbank-Relationen verwenden werden
  - für Queries

```
?- father(X,liz).

X = tom
```

  - zur Definition neuer Relationen
    - `greatgrandfather(X,Y) :- father(X,Z), grandparent(Z,Y).`
- Regeln werden auch “Klause(l)n” (“Clauses”) genannt.
  - **Horn-Klauseln:** Eine Klausel mit höchstens einem Literal im Head der Regel
    - theoretisch kann man auch allgemeinere Regeln definieren
- Zusammenhang mit Datenbanksystemen:
  - Das Definieren neuer Relationen entspricht in etwa der Definition eines Views auf eine Datenbank

Substitution

- **Ground Atoms:**
  - ein Atom, in dem keine Terme Variablen enthalten (= Fakten).

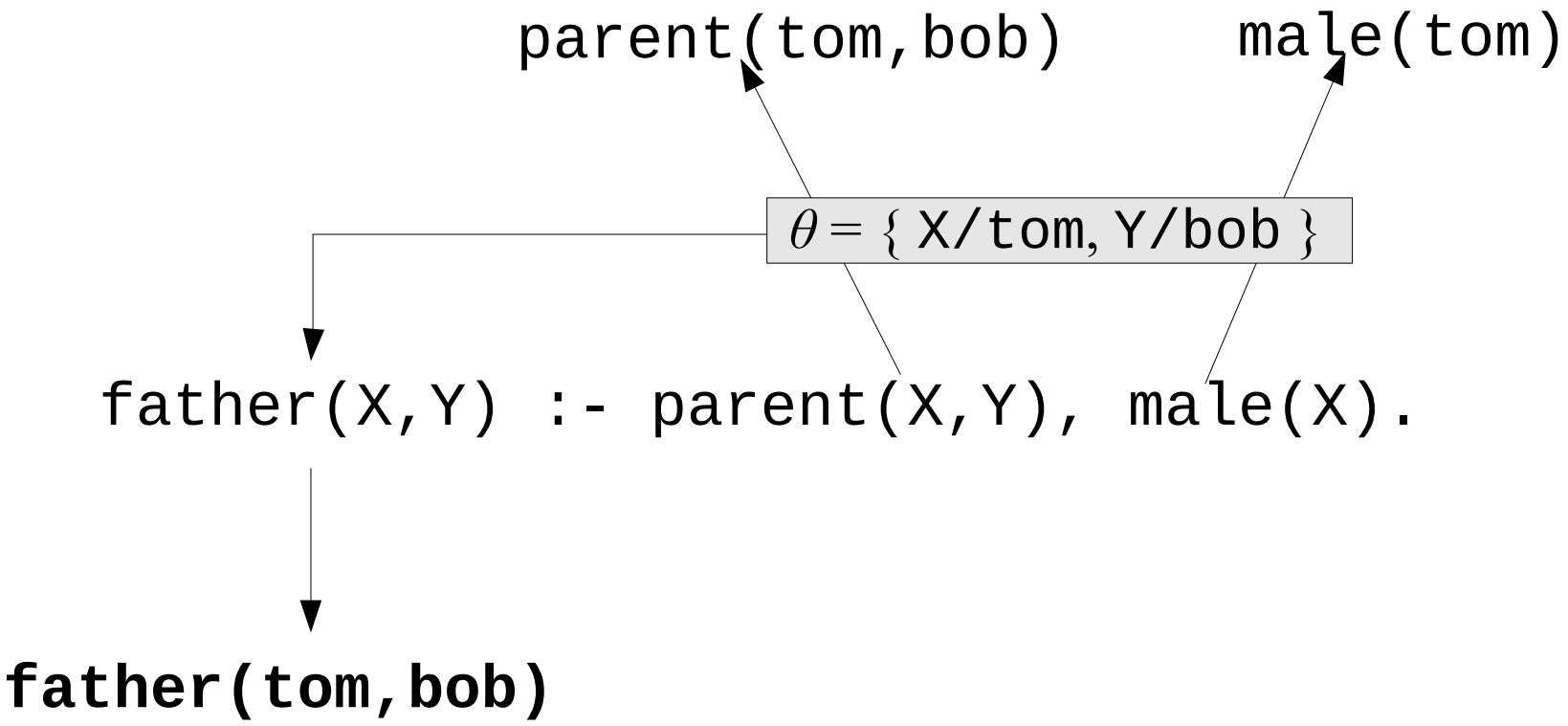
- Eine **Substitution** belegt Variablen eines Atoms mit einer Konstanten
  - z.B., um ein Atom in ein Ground Atom überzuführen.
- Beispiel:
  - Literal $L = \{$ `parent(X,Y)` $\}$
  - Substitution $\theta = \{$ `X/tom, Y/bob` $\}$
  - Anwendung der Substituion: $L\theta = \{$ `parent(tom,bob)` $\}$

- Eine Query versucht daher, alle gültigen Substitutionen zu finden
  - d.h., alle Substitution, die zu Fakten führen, die man in der Datenbank finden kann.

Beweisführung in Datalog

- Beweis = Ableitung neuer Fakten aus den definierten Programmen
- Elementary Production Principle (EPP)
  - Für jede Regel der Form H :- $B_1, ..., B_n$.
  - und eine Faktenmenge $F_1, ..., F_n$
  - und eine Substitution, die den Body der Regel auf die Menge der Fakten überführt $\theta : \forall i \in (1...n) : B_i\theta = F_i$
  - folgt in einem Beweisschritt aus dem Body das Literal, das sich aus der Anwendung von θ auf den Head ergibt
    - d.h., wir können $H\theta$ ableiten
- Man kann die Beweisschritte so lange iterieren, bis es keine Veränderung mehr gibt (Fixpunkt in der Faktenmenge)

Beispiel


parent(tom,bob)
male(tom)
θ = { X/tom, Y/bob }
father(X,Y) :- parent(X,Y), male(X).
father(tom,bob)

Wichtige Begriffe aus der Formalen Logik

- Atome können viele Interpretationen haben
  - Interpretation = Abbildung auf Entitäten in der reellen Welt
  - daher können verschiedene Aussagen wahr oder falsch sein
  - einige Aussagen werden jedoch immer wahr (bzw. immer falsch) sein
- Kanonische Interpretation:
  - Herbrand Universe:
    - die Menge aller Konstanten, die in den Fakten vorkommen
  - Herbrand Base:
    - die Menge aller Ground Atoms
      - also alle Aussagen, die sich aus den vorhanden Prädikaten-symbolen und dem Herbrand Universe bilden lassen
  - Herbrand Interpretation:
    - eine Untermenge der Herbrand Base, die die Menge alle *wahren* Aussagen in der Herbrand Base auszeichnet
  - Herbrand Model:
    - eine Herbrand Interpretation, in der alle Fakten und Regeln gelten

Fixpunkt-Semantik von Datalog

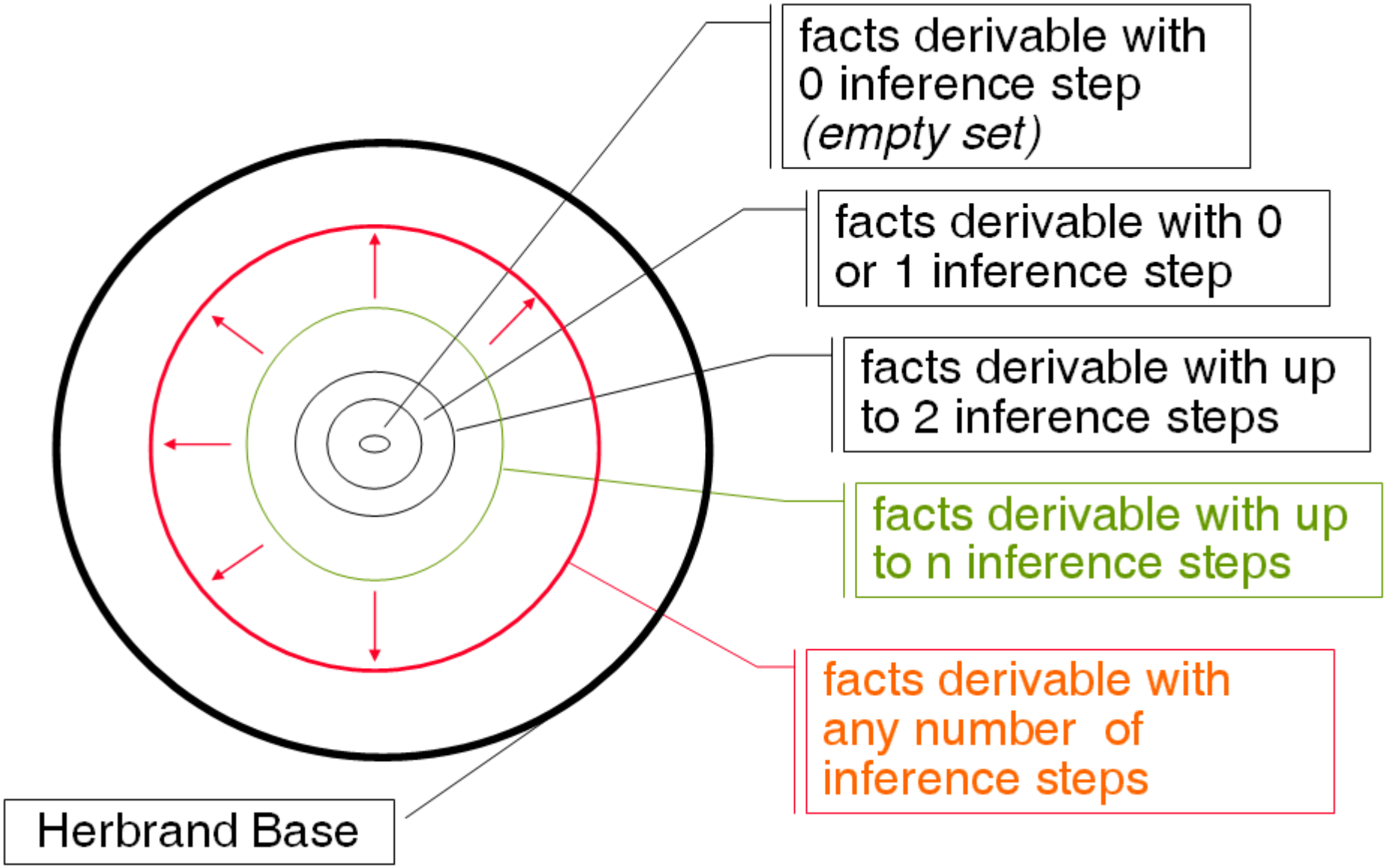


Figure taken from P. Gloess, Logic Programming, Jan. 2002

Beispielberechnung eines Fixpunkts

- Fakten: die übliche Menge `parent/2` und `male/1`
- Regeln:

```
father(X,Y) :- parent(X,Y), male(X).
grandfather(X,Y) :- father(X,Z), parent(Z,Y).
```

- 1. Iteration (alle Fakten):

```
parent(bob, ann), parent(bob, pat), parent(pam, bob),
parent(pat, jim), parent(tom, bob), parent(tom, liz),
male(bob), male(tom), male(jim),
```

- In der 2. Iteration kommen dazu:

```
father(bob, ann), father(bob, pat),
father(tom, bob), father(tom, liz),
```

- In der 3. Iteration kommen dazu:

```
grandfather(bob, jim), grandfather(tom, ann),
grandfather(tom, pat)
```

- In weiteren Iteration kann nichts Neues mehr abgeleitet werden → Fixpunkt gefunden.

Korrektheit von EPP

- Man kann beweisen, daß die Ableitungsregel EPP korrekt ist
  - Korrektheit = Konsistenz +Vollständigkeit
- Konsistenz
  - Es können mittels EPP keine logischen Widersprüche hergeleitet werden (also nicht sowohl A als auch $\neg$ A)
  - Aus einer falschen Aussage (wie z.B. A $\wedge \neg$ A) könnte man jede beliebige Aussage herleiten
    - *Ex falso quodlibet*: Wenn der Bedingungsteil einer Implikation falsch ist, ist die Implikation auf jeden Fall gültig
- Vollständigkeit
  - Alle Fakten, die logisch aus einer Menge von Regeln und Fakten folgern, können auch mittels EPP hergeleitet werden
    - alle wahren Fakten können bewiesen werden (u.U. mit mehreren Iterationen)

Regelmengen

- Es können auch mehrere Regeln zur Definition einer Relation verwendet werden
  - Diese Regeln werden dann ähnlich wie ein logisches ODER interpretiert
    - ein Fakt ist Teil der Relation, wenn es entweder die erste oder die zweite Regel erfüllt.
    - die Aneinander-Reihung mehrerer Klausen mit demselben Head entspricht eigentlich einer Disjunktion in der Definition
  - Beispiel:

```
person(X) :- male(X).
person(X) :- female(X).
```

    - X ist eine Person, wenn X entweder männlich oder weiblich ist
- Alle Definitionen zu einem Prädikat nennt man dann auch ein Datalog Programm

Rekursive Definitionen

- Ein Prädikat kann sowohl im Body als auch im Head einer Regel vorkommen
  → rekursives Programm

- Beispiel:

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

    - Jede Person ist ihr eigener Vorfahre
    - Wenn ein Vorfahre einen Elternteil hat, so ist dieser ebenfalls Vorfahre.
    ```

Fixpunkt-Berechnung für `ancestor/2`

- 1. Iteration: alle Fakten
- In der 2. Iteration kommen dazu:

```
person(bob), person(tom), person(jim),
person(pam), person(liz), person(pat), person(ann)
```

- In der 3. Iteration kommen dazu:

```
ancestor(bob,bob), ancestor(tom,tom), ancestor(jim,jim),
ancestor(pam,pam), ancestor(liz,liz), ancestor(pat,pat),
ancestor(ann,ann)
```

- In der 4. Iteration kommen dazu:

```
ancestor(bob,ann), ancestor(bob,pat), ancestor(pam,bob),
ancestor(pat,jim), ancestor(tom,bob), ancestor(tom,liz),
```

- In der 5. Iteration kommen dazu:

```
ancestor(pam,ann), ancestor(tom,ann), ancestor(pam,pat),
ancestor(tom,pat), ancestor(bob,jim)
```

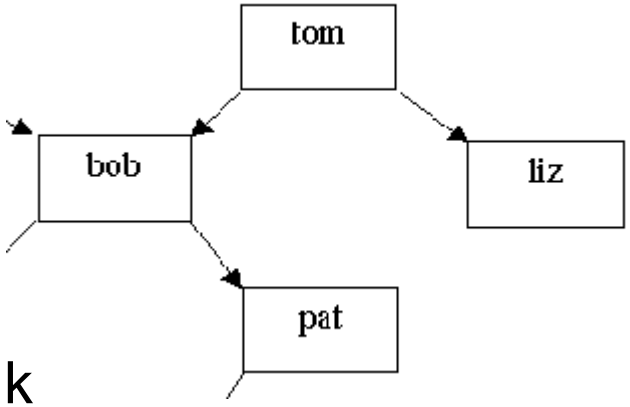
- In der 6. Iteration kommen dazu:

```
ancestor(pam,jim), ancestor(tom,jim)
```

→ Fixpunkt gefunden.

Negation

- Man kann auch Negationen formulieren
  - Beispiel: `\+ parent(liz,pat)`
- *Problem*:
  - Wir kennen einen Teil der Welt
    - z.B. die Eltern-Relationen in der Datenbank
  - aber nicht die gesamte Welt
    - in der Datenbank steht nicht, daß `liz` ein Elternteil von `pat` ist,
    - können wir daraus folgern daß sie es nicht ist?
- *Annahme*: Die Datenbank ist vollständig
  - **Closed World Assumption**:
    - Alles, was man nicht beweisen kann, ist falsch.
  - **Negation as failure**:
    - Die Verneinung einer Aussage wird als bewiesen angesehen, wenn man die Aussage nicht beweisen kann.



Probleme mit Negation

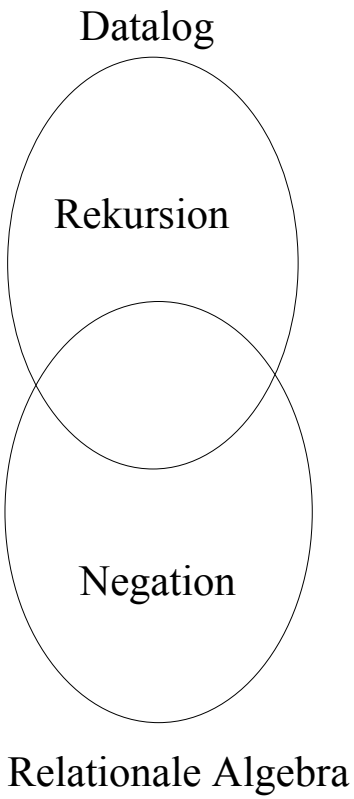
- Negation ist mit einfacher Fixpunkt-Semantik nicht in den Griff zu bekommen
  - Negation as Failure bedeutet ja, daß man die Negation annimmt, wenn man das nicht-negierte Literal nicht beweisen kann
  - Das weiß man aber erst am Ende!
    - Zu keinem Zeitpunkt der iterativen Anwendung von EPP kann man sicher sagen, daß ein Literal nicht noch später bewiesen wird
- Zusätzliche Mechanismen sind notwendig, um solche Fälle in den Griff zu bekommen
  - Stratified Datalog mit Negation:
    - negierte Literale werden zuerst mit CWA evaluiert
    - das funktioniert, solange es keine negierten rekursiven Aufrufe gibt (genauer gesagt: keine Zyklen, die eine Neg. enthalten)
    - problematisches Beispiel:

```
even(X)  :- \+ odd(X).
odd(X)   :- \+ even(X).
```

  - werden wir hier aber nicht weiter behandeln

Ausdrucksstärke von Datalog

- klassisches, nicht-rekursives Datalog ist gleich mächtig wie die Relationale Algebra ohne Differenz
  - um die Differenz zu definieren, braucht man Negation
- Datalog mit Negation ist gleichmächtig wie die Relationale Algebra
  - im Prinzip einfaches SQL `select-from-where`
- Datalog mit Negation und Rekursion ist mächtiger als Relationale Algebra
  - rekursive Datalog-Anfragen können nicht in SQL formuliert werden
- Beide sind jedoch nicht Turing-complete
  - aber Prolog ist Turing-complete



Relationale Algebra → Datalog

Die Operatoren der relationalen Algebra können leicht in Datalog formuliert werden

R = `r(X,Y,Z)`
S = `s(X,Y,Z)`

- Projektion: $\pi_2(R)$
  - `projection(Y) :- r(X,Y,Z).`
- Selektion: $\sigma_{X=c}(R)$
  - `selection(X,Y,Z) :- r(X,Y,Z), X=c.`
- Kartesisches Produkt: $R \times S$
  - `cartesian(X,Y,Z,A,B,C) :- r(X,Y,Z), s(A,B,C).`
- Union: $R \cup S$
  - `union(X,Y,Z) :- r(X,Y,Z).`
    `union(X,Y,Z) :- s(X,Y,Z).`
- Difference: $R - S$
  - `difference(X,Y,Z) :- r(X,Y,Z), not(s(X,Y,Z)).`
- (Natural) Join: $R \bowtie S$
  - `join(X,Y,Z,B,C) :- r(X,Y,Z), s(X,B,C).`

Datalog → Relationale Algebra

- Die meisten Datalog-Anfragen können in relationale Algebra übersetzt werden
  - Ausnahme: Rekursionen
- Ein allgemein-gültiger Algorithmus ist komplex
- Für einzelne Regeln funktioniert meistens:

1. Erzeuge für jedes Subgoal ein Schema, wobei die Attribute mit den Variablen-Namen benannt werden
2. Für ein negiertes Subgoal:
   a) Finde alle möglichen Variablen-Belegungen (Herbrand base)
   b) Subtrahiere davon alle Tupel, die das Goal erfüllen
3. Die Resultate von 1. und 2. werden mit Natural Joins zusammengefügt
4. Konstanten und Vergleiche implementiert man mit Selektionen
5. Das Resultat wird dann auf die Variablen des Heads projiziert

- Mehrere Regeln werden mit der Vereinigung verbunden

Based on a slide by J. Ullmann for Stanford course CS145, Fall 1997

```
grandfather(X,Y) :- parent(X,Z),
                    parent(Z,Y),
                    \+ grandmother(X,Y).
```

Beispiel

- berechne den Natural Join der ersten beiden Relationen
  - benenne sie *P1*(*x*,*z*) und *P2*(*z*,*y*) und speichere Resultat in *R1*(*x*,*z*,*y*)

$$R1(x,z,y) \leftarrow \rho_{P1(x,z)}(parent) \bowtie \rho_{P2(z,y)}(parent)$$

- berechne alle möglichen Tupel von *x* und *y*
  - das Kreuz-Produkt der Projektionen auf das erste und letzte Attribut von *R1*

$$R2(x,y) \leftarrow \pi_{x}(R1) \times \pi_{y}(R1)$$

- subtrahiere davon alle Tupel der `grandmother` relation
  - zuerst wieder umbennen

$$R3(x,y) \leftarrow R2 - \rho_{G(x,y)}(grandmother)$$

- berechne Natural Join des Resultats mit *R1*

$$grandfather(x,y) \leftarrow \pi_{x,y}(R1(x,z,y) \bowtie R3(x,y))$$

Based on a slide by J. Ullmann for Stanford course CS145, Fall 1997

Praktische Datalog Systeme

- Enge Integration mit Datenbank-Systemen
  - Extensional Database (EDB)
    - Relationales Datenbanksystem zur Speicherung der Relationen, die extensional, d.h. durch Auflisten aller Tupel, definiert werden
  - Intensional Database (IDB)
    - Alle anderen Relationen, die durch andere Relationen definiert werden (d.h., Regeln und Programme)
- Optimierung der Beweisführung
  - Naive Berechnung des Fixpunkts zu ineffizient
  - Effizientere Methoden
    - konzentrieren sich nur auf Fakten, die in der letzten Iteration neu hinzugekommen sind.
    - versuchen eine Query umzuformulieren, sodaß sie effizienter berechnet werden kann (Elimination von Redundanzen)
  - Details siehe z.B. (Ceri, Gottlob, Tanza, IEEE-KDE, 1989)

Deklarative Semantik

- Datalog hat eine rein deklarative Semantik
- Fokus ist auf dem Finden aller Lösungen
  - **Forward Chaining**
    - man geht von den Fakten bzw. dem Body der Regel aus
  - **Breadth-First Search**
    - man findet zuerst alle Ableitungen in einem Schritt, dann in zwei Schritten, etc.
- Die Interpretation der Klausen erfolgt nach rein logischen Gesichtspunkten
  - Die Klausen bestimmen die Menge der Tupel, die wahr sein sollen
  - Ihre Reihenfolge ist egal, genauso wie die Reihenfolge der Bedingungen in einer Regel egal ist.

Prozedurale Semantik

- Prolog hat eine prozedurale Semantik
- Fokus liegt auf dem Finden einer Lösung
- Es gibt strenge Vorschriften, wie Prolog-Programme ausgeführt werden müssen
  - **Backward Chaining**
    - man geht vom Goal bzw. dem Head der Regel aus
  - **Depth-First Suche**
    - man versucht möglichst schnell eine Substitution zu finden
  - **Backtracking**
    - Wenn sich eine Substitution in der Folge als nicht erfüllbar herausstellt, wird der letzte Schritt zurückgenommen und eine alternative Substitution versucht
  - **Reihenfolge** der Abarbeitung der Klauseln und Literale ist fix festgelegt
    - von oben nach unten, von links nach rechts
  - **System-Prädikate** wie Cut

Prolog vs. Datalog

- Einfache Kopplung von Prolog mit Datenbank-Systemen
  - EDB: Die Datenbank übernimmt die Speicherung aller Fakten
  - IDB: Prolog übernimmt die Regeln und das automatische Schließen
- Prolog ist als Datenbankanfrage-Sprache nicht besonders gut geeignet
  - die Prozedurale Semantik ist für Anfragen der Art "*Gibt es eine Substitution, die dieses Literal wahr macht?*" optimiert
  - das ist recht ineffizient im Vergleich zu den Mengen-orientierten Techniken, die von optimierten Datenbank-Systemen verwendet werden
- Prolog ist hingegen eine vollwertige Programmiersprache mit Konstrukten für Rekursionen, Iterationen, Selektionen, etc.
  - in der KI in Europa und Japan recht verbreitet

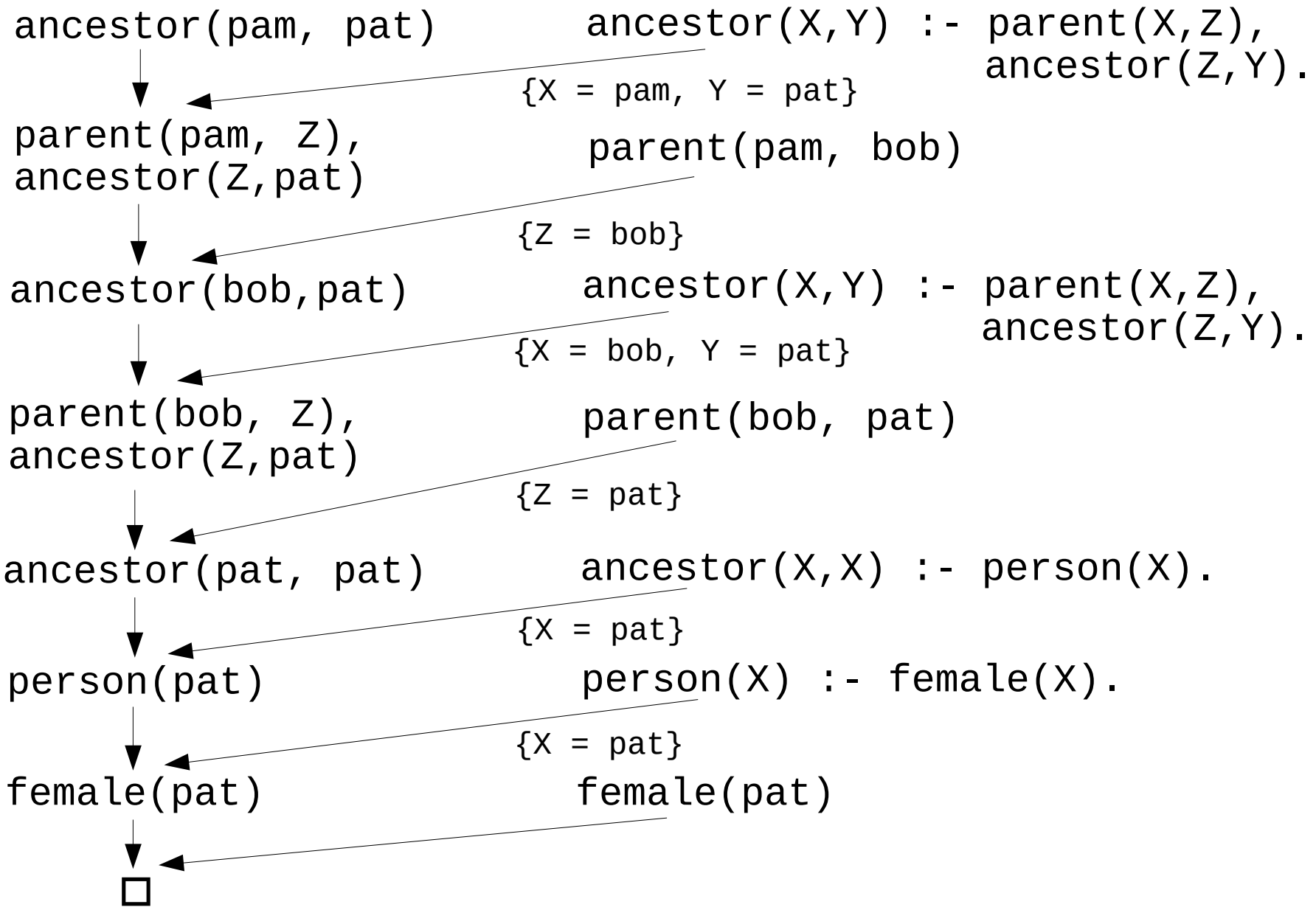
Resolution

- Resolutionsprinzip
  - Um eine Aussage zu beweisen, nimmt man ihr Gegenteil an, und führt das zu einem Widerspruch.
  - Daher eignet sich Resolution gut, um einzelne Aussagen zu beweisen
    - terminiert nachdem eine Lösung gefunden wurde
    - bzw. Eingabe eines Strichpunkts, um die nächste Lösung zu finden (forciertes Backtracking)
- Resolution ist
  - konsistent
    - nur wahre Fakten werden hergeleitet
  - *nicht* vollständig
    - gewisse logisch wahre Sätze wie `a :- a` können nicht hergeleitet werden (Tautologien)
  - aber refutations-vollständig (refutation-complete)
    - jedes wahre Goal/Fakt kann bewiesen werden (bzw. sein Gegenteil auf einen Widerspruch geführt werden)

SLD-Resolution

- SLD-resolution
  - Linear resolution for Definite clauses with Selection function
- Selection Function
  - Prolog wählt immer die erste Regel, die mit dem momentanen Goal matcht
  - Um mehrere Goals zu beweisen (z.B. den Body einer Regel), werden die Literale von links nach rechts abgearbeitet
- Lineare resolution
  - Es wird immer das Ergebnis des letzten Ableitungsschrittes weiterverarbeitet
  - d.h. der nächste Schritt beginnt mit dem letzten Resultat
- Definite Clauses
  - Regeln, die im Head genau ein Literal stehen haben
  - In Prolog gibt es nur definite clauses

SLD Beweisbaum



```
?- ancestor(pam,pat).
 T Call: (7) ancestor(pam, pat)
 T Call: (8) parent(pam, Z)
 T Exit: (8) parent(pam, bob)
 T Call: (8) ancestor(bob, pat)
 T Call: (9) parent(bob, Z)
 T Exit: (9) parent(bob, ann)
 T Call: (9) ancestor(ann, pat)
 T Call: (10) parent(ann, Z)
 T Fail: (10) parent(ann, Z)
 T Fail: (9) ancestor(ann, pat)
 T Redo: (9) parent(bob, Z)
 T Exit: (9) parent(bob, pat)
 T Call: (9) ancestor(pat, pat)
 T Call: (10) person(pat)
 T Call: (11) male(pat)
 T Fail: (11) male(pat)
 T Redo: (10) person(pat)
 T Call: (11) female(pat)
 T Exit: (11) female(pat)
 T Exit: (10) person(pat)
 T Exit: (9) ancestor(pat, pat)
 T Exit: (8) ancestor(bob, pat)
 T Exit: (7) ancestor(pam, pat)
Yes
```

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

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

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

```
ancestor(X,X) :- person(X).
```

```
person(X) :- male(X).
```

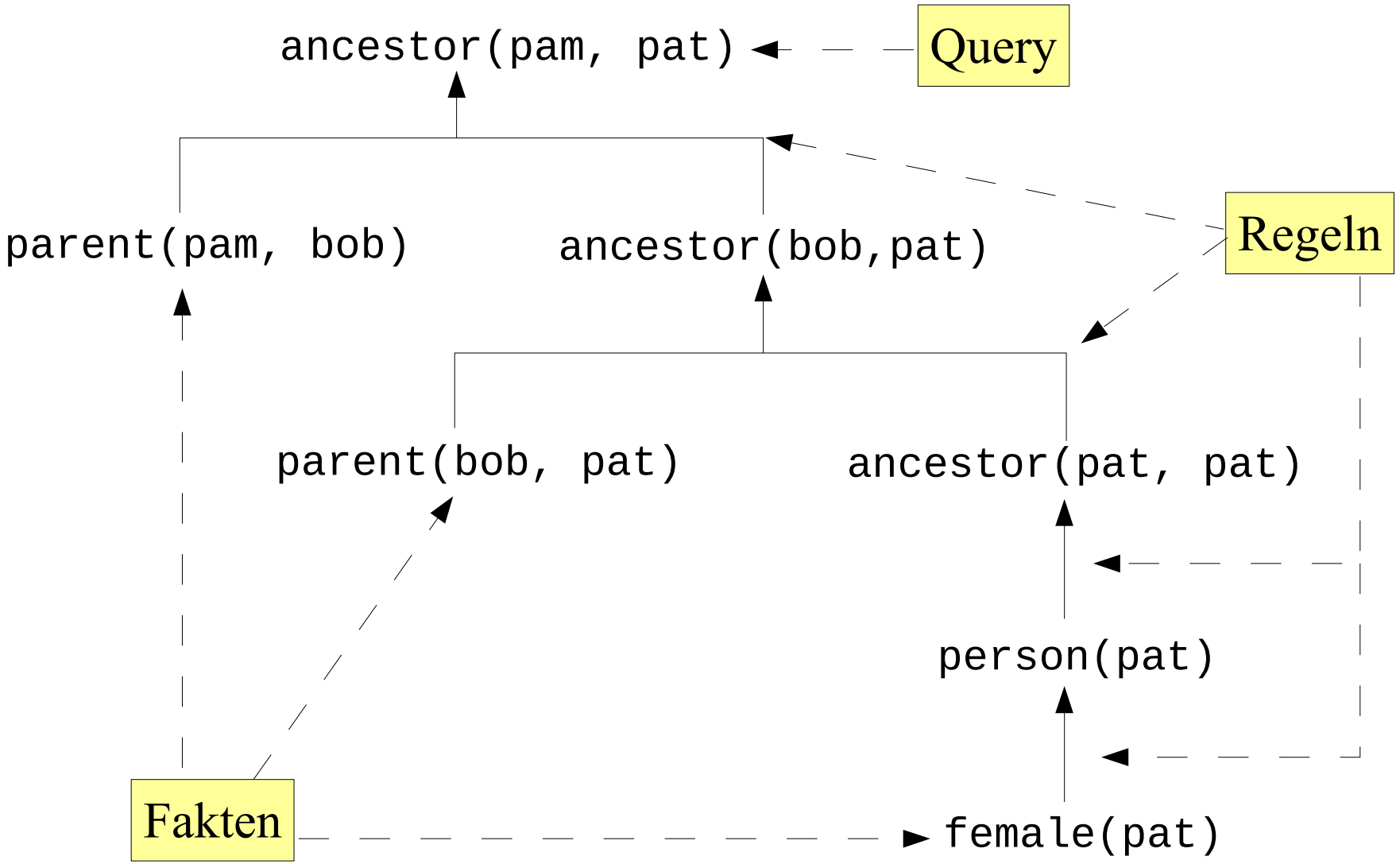
```
person(X) :- female(X).
```

SLD Suchbaum

zeigt den Ablauf des Beweises

Generischer Beweisbaum

- zeigt, wie eine bewiesene Query aus den Fakten folgt



Cut (!) and fail in Prolog

- `!` (Cut) und `fail`
  - `!` verhindert Backtracking
  - `fail` läßt Goal fehlschlagen

Beispiel: Definition von `not`

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

- Beispiel-Query: `not(parent(liz,ann)).`
  - Versuch, das Goal mit der ersten Regel zu beweisen:
    - `X = parent(liz,ann)`
    - Es wird versucht, `X` zu beweisen.
    - Kein Beweis gefunden → Subgoal schlägt fehl
  - Versuch, das Goal mit der zweiten Regel zu beweisen:
    - Die Regel hat keine Bedingung (d.h. sie feuert immer)

→ `not(parent(liz,ann))` ist bewiesen.

Cut (!) and fail in Prolog

- `!` (Cut) und `fail`
  - `!` verhindert Backtracking
  - `fail` läßt Goal fehlschlagen

Beispiel: Definition von `not`

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

- Beispiel-Query: `not(parent(bob,ann)).`
  - Versuch, das Goal mit der ersten Regel zu beweisen:
    - `X = parent(bob,ann)`
      - `X` ist ein Fakt → Subgoal bewiesen
    - `!` evaluiert zu true
    - `fail` schägt fehl
      - Kein Beweis gefunden → Subgoal schlägt fehl
    - `!` verhindert Backtracking, d.h. es werden keine alternativen Beweiswege mehr gesucht (z.B. Beweis mit Hilfe der 2. Regel)

→ `not(parent(bob,ann))` kann nicht bewiesen werden

- kann daher als falsch angesehen werden

Weitere Prolog-Erweiterungen

- Funktionen
  - Terme können nicht nur Konstanten und Variablen sein, sondern auch Funktionssymbole.
  - **Achtung:** Funktionen sind nur Schreibweise, werden nicht definiert oder evaluiert!
    - haben aber spezielle Regeln beim Matchen (Unifikation)
    - dienen zum Aufbau komplexerer Datenstrukturen

- Listen
  - Schreibweise: `[Head|Body]`
    `[a|[b|[c|[]]]] = [a,b,c]` (Kurzschreibweise)
  - sind eigentlich verschachtelte Funktionen
    - Beispiel: (`cons` sei ein Funktionssymbol für Listen, `nil` für die leere Liste)

```
member(a,[a,b,c]) = member(a,cons(a,cons(b,cons(c,nil))))
```

Beispiel-Programme in Prolog

- member/2

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

- append/3

```
append([X|Xs],Ys,[X|Zs]) :-
   append(Xs,Ys,Zs).
append([],Xs,Xs).
```

- reverse/2

```
reverse(X,Y) :-
   reverse(X,[],Y).

reverse([],X,X).
reverse([X|Xs],Y,Z) :-
   reverse(Xs,[X|Y],Z).
```

- union/3

```
union([],A,A) :- !.
union([A|B],C,D) :-
        member(A,C), !,
        union(B,C,D).
union([A|B],C,[A|D]) :-
        union(B, C, D).
```

Diese und ähnliche Prädikate sind in den meisten Prolog-Systemen ebenfalls fix eingebaut

Beispiel-Programm member/2

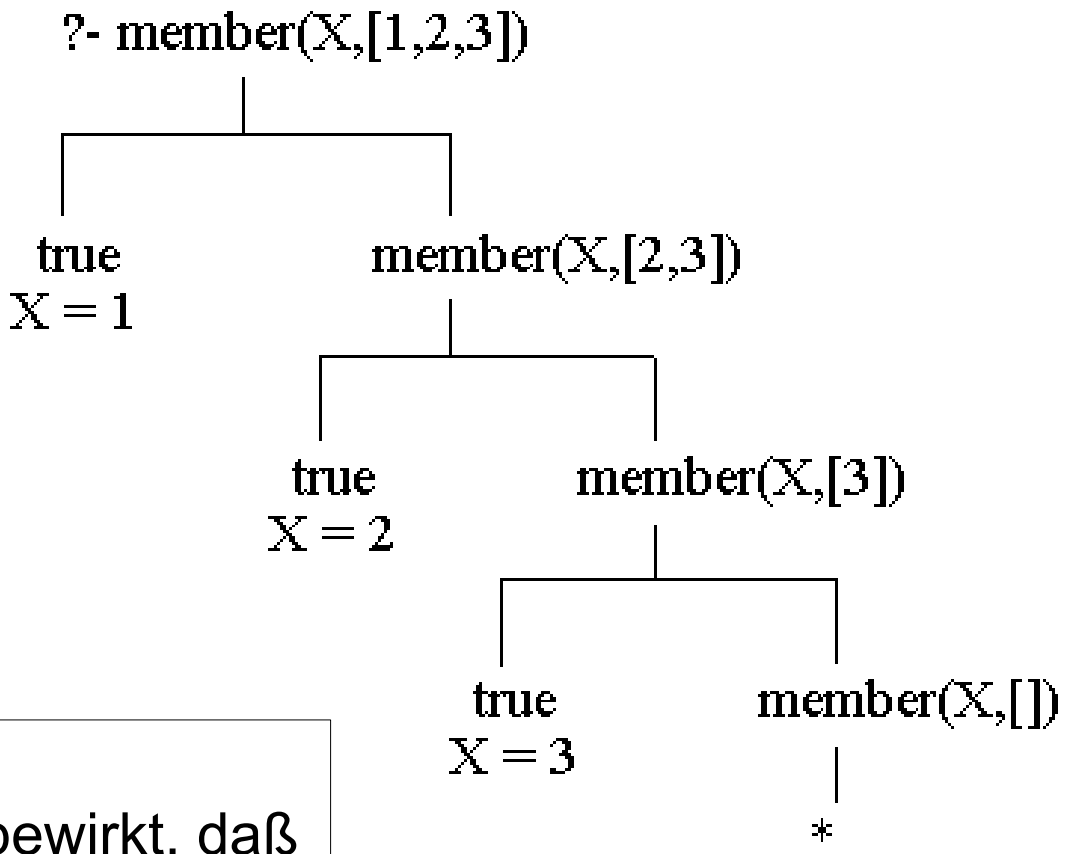
- Programm

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

- Query

```
?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
No
```

- Abarbeitung der Query



Erinnerung:
`;` ist eine Benutzereingabe, die bewirkt, daß nach der nächsten Lösung gesucht wird

Eingebaute Prädikate

- `assert/1` fügt der Wissensbasis ein Fakt hinzu
  - z.B. `assert(parent(liz,bill)).`
- `retract/1` entfernt ein Fakt aus der Wissensbasis
  - z.B. `retract(parent(liz,bill)).`
- `copy_term/2` kopiert einen Term mit neuen Variablen
  - z.B.

```
?- copy_term(meinTerm(X,Y),Copy).

X = _G157
Y = _G158
Copy = meinTerm(_G249, _G250)
```

Neue, vom System vergebene Variablen-Namen

  - Da bei `retract/1` alle Variablen in einem Term/einer Klause immer neu ersetzt werden, ist eine einfache Implementierung:

```
copy_term(Term,Copy) :-
        assert(dummy_for_copy(Term)),
        retract(dummy_for_copy(Copy)).
```

Eingebaute Prädikate

- `clause/2`
  - überprüft, ob sich eine Clause mit einem gegebenen Head in der Wissensbasis befindet, und retourniert den Body
  - z.B.

```
?- clause(person(X),Body).

X = _G157
Body = male(_G157)
```

  - Fakten können ebenfalls mit clause/2 überprüft werden, dann ist der Body true.
  - z.B.

```
?- clause(parent(bob,pat),Body).

Body = true
```

Prolog Meta-Interpreter

- Es ist einfach, ein Programm zu schreiben, das PROLOG selbst interpretiert
- Beispiel eines einfachen Prolog-(Meta-)Interpreters:

```
% Beweis eines Goals durch ein Fakt
prove( L )          :- clause( L, true).

% Beweis einer Konjunktion von Goals
prove( (C1,C2) ) :- !,
                    prove(C1),
                    prove(C2).

% Beweis eines Goals mithilfe einer Regel
prove( Head )       :- clause( Head, Body ),
                    Body \= true,
                    prove(Body).
```

Der Cut wird benötigt, da `clause((C1,C2),(C1,C2))` gilt, und bei der nächsten Regel zu einer infiniten Rekursion führen würde.

Meta-Interpreter

- Mit einem Meta-Interpreter läßt sich das Verhalten der Implementation beliebig ändern
  - einfaches Beispiel: Die Auswertungsreihenfolge der UND-Verknüpfung vertauschen:

```
% Beweis einer Konjunktion
% (aber von rechts nach links)
prove( (C1,C2) ) :- !,
                    prove(C2),
                    prove(C1).
```

- Man könnte z.B.
  - Breadth-First Suche statt Depth-First Suche implementieren
  - Forward Chaining statt Backward Chaining
  - Debugging Informationen (z.B. Beweisbäume) erzeugen
  - u.v.m.
- So ist z.B. DES Datalog ein komplexer PROLOG-Meta-Interpreter

Beispiel: Beweisbäume erzeugen

```
% Der Beweis eines Fakts ist das Fakt selbst
proof_tree( L, L) :-
     clause(L, true).

% Der Beweis einer Konjunktion von Goals ist die
% Konjunktion der Beweise für die Goals
proof_tree( (C1,C2), (ProofC1, ProofC2) ) :-
     !,
     proof_tree(C1, ProofC1),
     proof_tree(C2, ProofC2).

% Der Beweis eines Goals mit einer Regel ist der
% Beweis des Bodies
proof_tree( Head, (Head :- BodyProof))  :-
     clause( Head , Body ),
     Body \= true,
     proof_tree(Body, BodyProof).
```

Beispiel

- Beispiel-Ausgabe

```
?- proof_tree(ancestor(pam,pat),P).

P = ancestor(pam, pat) :-
        parent(pam, bob),
        (ancestor(bob, pat) :-
            parent(bob, pat),
            (ancestor(pat, pat) :-
                (person(pat) :-
                    female(pat)
                )
            )
        )
```

Anm: die Ausgabe wurde hier zur besseren Lesbarkeit händisch formatiert