Logische Programmierung

im Rahmen der LV

Logik und Logikprogrammierung

apl. Prof. Dr.-Ing. habil.

Rainer Knauf

Fachgebiet Künstliche Intelligenz

Fakultät für Informatik & Automatisierung

Technische Universität Ilmenau

Zuse-Bau, Raum 3060 (Sekretariat)

Tel. 03677 69-1445, 0361 3733867, 0172 9418642

rainer.knauf@tu-ilmenau.de

Auf der Seite

http://www.tu-ilmenau.de/ki/lehre/

sind downloadbar:

1. Skript zur Vorlesung
2. diese Foliensammlung als PowerPoint-Präsentation
3. Übungsaufgaben zum Seminar
4. Entwicklungsumgebung für Prolog Programme ***Visual Prolog Personal Edition®***
5. Anleitung und Aufgabenstellung für ein Praktikum (wer das wünscht) inkl. Hinweise zur Praktikumsdurchführung und Testumgebung zum Testen der gefundenen Lösung

Inhalt:

1. Einführung
2. Erinnerung und Anpassung von Termini: Grundlagen der Logischen Programmierung
   - Prädikatenkalkül der ersten Stufe (PK1)
   - Deduktion im PK1: Resolutionsmethode, Unifikation
3. Logische Programmierung
   - Einordnung des logischen Programmierparadigmas
   - Syntax logischer Programme
   - PROLOG aus logischer Sicht
   - PROLOG aus prozeduraler Sicht
   - Listen, (Links- und Rechts-) Rekursion, Akkumulator-Variablen, Differenzlistentechnik
   - Typische Problemklassen der Anwendung Logischer Programmierung

1 Einführung in die Künstliche Intelligenz (KI)

Ziel : Mechanisierung von Denkprozessen

Grundidee *(nach G.W. Leibniz)*

1. lingua characteristica — Wissensdarstellungssprache
2. calculus ratiocinator — Wissensverarbeitungskalkül

Teilgebiete der KI

- Wissensrepräsentation
- maschinelles Beweisen (Deduktion)
- KI-Sprachen: Prolog, Lisp
- Wissensbasierte Systeme
- Lernen (Induktion)
- Wissensverarbeitungstechnologien (Suchtechniken, fallbasiertes Schließen, Multiagenten-Systeme)
- Sprach- und Bildverarbeitung

2 Logische Grundlagen

2.1 PROLOG – ein „Folgerungstool“

Sei ***M*** eine Menge von Aussagen, ***H*** eine Hypothese.

H folgt aus M (***M*** |= ***H***) , falls jede Interpretation, die zugleich alle Elemente aus ***M*** wahr macht (jedes Modell von M), auch ***H*** wahr macht.

Für endliche Aussagenmengen $M = \{A_1, A_2, \ldots, A_n\}$ bedeutet das:

M |= ***H*** , gdw. $ag(\bigwedge_{i=1}^{n} A_i \rightarrow H)$

bzw. (was dasselbe ist) $kt(\bigwedge_{i=1}^{n} A_i \wedge \neg H)$

2.2 Aussagen in PROLOG: HORN-Klauseln des PK1

$$\forall X_1 \ldots \forall X_n \left(\underbrace{A(X_1, \ldots X_n)}_{Klauselkopf} \leftarrow \underbrace{\bigwedge_{i=1}^{m} A_i(X_1, \ldots X_n)}_{Klauselkörper} \right)$$

$A(X_1,..., X_n)$, $A_i(X_1,..., X_n)$
quantorfreie Atomformeln, welche die allquantifizierten Variablen $X_1,..., X_n$ enthalten können

Varianten / Spezialfälle

1. **Regeln** (vollständige HORN-Klauseln)

$$\forall X_1 ... \forall X_n (A(X_1,...,X_n) \leftarrow \bigwedge_{i=1}^{m} A_i(X_1,...,X_n))$$

2. **Fakten** (HORN-Klauseln mit leerem Klauselkörper)

$$\forall X_1 ... \forall X_n (A(X_1,...,X_n) \leftarrow true)$$

3. **Fragen** (HORN-Klauseln mit leerem Klauselkopf)

$$\forall X_1 ... \forall X_n (false \leftarrow \bigwedge_{i=1}^{m} A_i(X_1,...,X_n))$$

4. **leere HORN-Klauseln** (mit leeren Kopf & leerem Körper)

$$false \leftarrow true$$

Effekte der Beschränkung auf HORN-Logik

1. Über HORN-Klauseln gibt es ein korrektes und vollständiges Ableitungsverfahren.
   - $\{K_1, ..., K_n\} \models H$, gdw. $\{K_1, ..., K_n\} \vdash_{ROB} H$

2. Die Suche nach einer Folge von Resolutionsschritten ist algorithmisierbar.
   - Das Verfahren „Tiefensuche mit Backtrack“ sucht systematisch eine Folge, die zur leeren Klausel führt.
   - Rekursive und/oder metalogische Prädikate stellen dabei die Vollständigkeit in Frage.

3. Eine Menge von HORN-Klauseln mit nichtleeren Klauselköpfen ist stets erfüllbar; es lassen sich keine Widersprüche formulieren.
   - $K_1 \wedge K_2 \wedge ... \wedge K_n \neq$ false

Die systematische Erzeugung von (HORN-) Klauseln

1. Verneinungstechnischen Normalform (VTNF): $\neg$ steht nur vor Atomformeln

$$\neg\neg A \equiv A$$
$$\neg (A \wedge B) \equiv \neg A \vee \neg B$$
$$\neg (A \vee B) \equiv \neg A \wedge \neg B$$
$$\neg (A \rightarrow B) \equiv A \wedge \neg B$$
$$\neg (A \leftrightarrow B) \equiv (A \wedge \neg B) \vee (\neg A \wedge B)$$
$$\neg \forall X\, A(X) \equiv \exists X \neg A(X)$$
$$\neg \exists X\, A(X) \equiv \forall X \neg A(X)$$

2. Erzeugung der Pränexen Normalform (PNF): $\forall$, $\exists$ stehen vor dem Gesamtausdruck

$$\nabla X\, A(X) \circ B \equiv \nabla X\, (A(X) \circ B)$$

- *falls X nicht in B vorkommt*
- $\nabla \in \{ \forall, \exists \}$, $\circ \in \{ \wedge, \vee \}$

$$A \rightarrow \nabla X\, B(X) \equiv \nabla X\, (A \rightarrow B(X))$$
$$\forall X\, A(X) \rightarrow B \equiv \exists X\, (A(X) \rightarrow B)$$
$$\exists X\, A(X) \rightarrow B \equiv \forall X\, (A(X) \rightarrow B)$$

3. Erzeugung der SKOLEM‘schen Normalform (SNF): $\exists$ wird eliminiert

Notation aller existenzquantifizierten Variablen als Funktion derjenigen allquantifizierten Variablen, in deren Wirkungsbereich ihr Quantor steht.
Dies ist keine äquivalente - , wohl aber eine die Kontradiktorizität erhaltende Umformung.

4. Erzeugung der Konjunktiven Normalform (KNF)

Durch systematische Anwendung des Distributivgesetzes

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

lässt sich aus der SNF $\forall X_1 \ldots \forall X_n\, A(X_1, \ldots, X_n)$ *stets die äquivalente KNF*

$$\forall X_1 \ldots \forall X_n\, ((L_1^{1} \vee \ldots \vee L_1^{n_1}) \wedge \ldots \wedge (L_m^{1} \vee \ldots \vee L_m^{n_m}))$$

erzeugen. Die L_i^k *sind unnegierte oder negierte Atomformeln und heißen positive bzw. negative Literale.*

5. Erzeugung der Klauselform (KF)

Jede der Elementardisjunktionen $(L_i^j \vee \ldots \vee L_i^{jk})$ *der KNF kann man als äquivalente Implikation (Klausel)*

$$(L_i^j \vee \ldots \vee L_i^{jm}) \leftarrow (L_i^{jm+1} \wedge \ldots \wedge L_i^{jk})$$

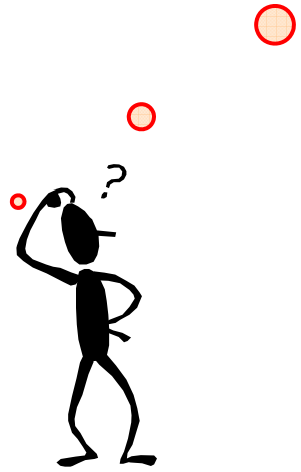
notieren, indem man alle positiven Literale $L_i^j, \ldots, L_i^{jm}$ *disjunktiv verknüpft in den* DANN*-Teil (Klauselkopf) und alle negativen Literale* $L_i^{jm+1}, \ldots, L_i^{jk}$ *konjunktiv verknüpft in den* WENN*-Teil (Klauselkörper) notiert.*

6. Sind die Klauseln aus Schritt # 5 HORN ?

In dem Spezialfall, dass alle Klauselköpfe dabei aus genau einem Literal bestehen, war die systematische Erzeugung von HORN-Klauseln erfolgreich; anderenfalls gelingt sie auch nicht durch andere Verfahren.

Heißt das etwa, die HORN-Logik ist eine echte Beschränkung der Ausdrucksfähigkeit ?

Richtig, das heißt es.



☺ *Im Logik-Teil dieser Vorlesung lernten Sie eine Resolutionsmethode für Klauseln kennenlernen*

☹ *... deren Algorithmisierbarkeit allerdings an der „kombinatorischen Explosion“ der Resolutionsmöglichkeiten scheitert, aber …*

☺ *… in der LV „Inferenzmethoden“ können Sie noch ein paar „Tricks“ kennenlernen, die „Explosion“ einzudämmen*

2.3 Inferenz in PROLOG: Resolution nach ROBINSON

gegeben:

- Menge von Regeln und Fakten M
- negierte Hypothese $\neg H$

$$M = \{K_1, ..., K_n\}$$

$$\neg H \equiv \neg \bigwedge_{i=1}^{m} H_i \equiv false \leftarrow \bigwedge_{i=1}^{m} H_i$$

Ziel:

- Beweis, dass $M \models H$

$$kt(\bigwedge_{i=1}^{n} K_i \wedge \neg H)$$

Eine der Klauseln habe die Form $A \leftarrow \bigwedge_{k=1}^{p} B_k$. (A, B_k – Atomformeln)

Es gebe eine Substitution (Variablenersetzung) ϑ für die in A und eines der H_i (etwa H_l) vorkommenden Variablen, welche A und H_l syntaktisch identisch macht.

$$M' \equiv \bigwedge_{i=1}^{n} K_i \wedge \neg(\underbrace{\bigwedge_{i=1}^{m} H_i}_{H})$$

ist kontradiktorisch (*kt M‘*), gdw. *M‘* nach Ersetzen von *H* durch

$$\bigwedge_{i=1}^{l-1} \vartheta\,(H_i) \wedge \bigwedge_{k=1}^{p} \vartheta\,(B_k) \wedge \bigwedge_{i=l+1}^{m} \vartheta\,(H_i)$$

noch immer kontradiktorisch ist.

Na „prima“!?
Jetzt wissen wir also, wie man die zu zeigende Kontradiktorizität auf eine andere – viel kompliziertere (?) – Kontradiktorizität zurückführen kann.

Für $p=0$ und $m=1$ wird es allerdings trivial.

Die sukzessive Anwendung von Resolutionen muss diesen Trivialfall systematisch herbeiführen:

Satz von ROBINSON

$$M' \equiv \bigwedge_{i=1}^{n} K_i \wedge \neg H$$

ist kontradiktorisch (*kt M'*), gdw. durch wiederholte Resolutionen in endlich vielen Schritten die negierte Hypothese $\neg H \equiv false \leftarrow H$ durch die leere Klausel $false \leftarrow true \equiv$ ersetzt werden kann.

Substitution

- Eine (Variablen-) Substitution ϑ einer Atomformel ***A*** ist eine Abbildung der Menge der in ***A*** vorkommenden Variablen ***X*** in die Menge der Terme (aller Art: Konstanten, Variablen, strukturierte Terme).
- Sie kann als Menge von Paaren *[Variable,Ersetzung]* notiert werden:
  $\vartheta = \{[x,t]: x \in X, t = \vartheta(x)\}$
- Für strukturierte Terme wird die Substitution auf deren Komponenten angewandt:
  $\vartheta(f(t_1, ..., t_n)) = f(\vartheta(t_1), ... , \vartheta(t_n))$
- Verkettungsoperator $\circ$ für Substitutionen drückt Hintereinander-anwendung aus:
  $\delta \circ \vartheta(t) = \delta(\vartheta(t))$
- Substitutionen, die zwei Terme syntaktisch identisch machen, heißen **Unifikator**:
  ϑ unifiziert zwei Atomformeln (oder Terme) ***s*** und ***t*** (oder: heißt Unifikator von ***s*** und ***t***), falls dessen Einsetzung **s** und **t** syntaktisch identisch macht.

Unifikation

Zwei **Atomformeln** $p_1(t_{11}, \ldots, t_{1n})$ und $p_2(t_{21}, \ldots, t_{2m})$ sind **unifizierbar**, gdw.

- sie die gleichen Prädikatensymbole aufweisen ($p_1 = p_2$) ,
- sie die gleichen Stelligkeiten aufweisen ($n = m$) und
- die Terme t_{1i} und t_{2i} jeweils miteinander unifizierbar sind.

Die Unifizierbarkeit zweier **Terme** richtet sich nach deren Sorte:

1. Zwei Konstanten t_1 und t_2 sind unifizierbar, gdw. $t_1 = t_2$.
2. Zwei strukturierte Terme $f_1(t_{11}, \ldots, t_{1n})$ und $f_2(t_{21}, \ldots, t_{2m})$ sind unifizierbar, gdw.
   - sie die gleichen Funktionssymbole aufweisen ($f_1 = f_2$) ,
   - sie die gleichen Stelligkeiten aufweisen ($n = m$) und
   - die Terme t_{1i} und t_{2i} jeweils miteinander unifizierbar sind.
3. Eine Variable t_1 ist mit einer Konstanten oder einem strukturierten Term t_2 unifizierbar. t_1 wird durch t_2 ersetzt (instanziert): $t_1 := t_2$.
4. Zwei Variablen t_1 und t_2 sind unifizierbar und werden gleichgesetzt: $t_1 := t_2$ bzw. $t_2 := t_1$.

Genügt „irgendein“ Unifikator?

‘n Beispiel

$K_1 : p(A,B) \leftarrow q(A) \wedge r(B)$

$K_2 : q(c) \leftarrow true$

$K_3 : r(d) \leftarrow true$

$\neg H : false \leftarrow \underbrace{p(X,Y)}_{H_1^0}$

Unifikatoren für H_1^0 und Kopf von K_1:

$\vartheta^1 = \{[X,a],[Y,b],[A,a],[B,b]\}$

$\vartheta^2 = \{[X,a],[Y,B],[A,a]\}$

$\vartheta^3 = \{[X,A],[Y,b],[B,b]\}$

$\vartheta^4 = \{[A,X],[B,Y]\}$

… *u.v.a.m.*

- Obwohl $\{K_1, K_2, K_3\} \models H$, gibt es bei Einsetzung von ϑ^1, ϑ^2, und ϑ^3 keine Folge von Resolutionsschritten, die zur leeren Klausel führt.
- Bei Einsetzung von ϑ^4 hingegen gibt es eine solche Folge.
- Die Vollständigkeit des Inferenz-verfahrens hängt von der Wahl des „richtigen“ Unifikators ab.
- Dieser Unifikator muss möglichst viele Variablen variabel belassen. Unnötige Spezialisierungen versperren zukünftige Inferenzschritte.

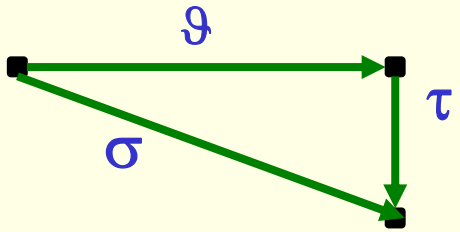
Ein solcher Unifikator heißt **allgemeinster Unifikator** bzw. most general unifier (*m.g.u.*).

Allgemeinster Unifikator

Eine Substitution ϑ heißt allgemeinster Unifikator (most general unfier ***m.g.u.***) zweier (Atomformeln oder) Terme ***s*** und ***t*** ($\vartheta = m.g.u.(s, t)$), gdw.

1. die Substitution ϑ ein Unifikator von ***s*** und ***t*** ist und
2. für jeden anderen Unifikator σ von ***s*** und ***t*** eine nichtleere und nicht identische Substitution τ existiert, so dass $\sigma = \tau \circ \vartheta$ ist.

graphisch betrachtet:



*Der Algorithmus zur Berechnung des **m.g.u.** zweier Terme **s** und **t** verwendet* **Unterscheidungsterme**:

Man lese ***s*** und ***t*** zeichenweise simultan von links nach rechts. Am ersten Zeichen, bei welchem sich ***s*** und ***t*** unterscheiden, beginnen die Unterscheidungsterme ***s**** und ***t**** und umfassen die dort beginnenden (vollständigen) Teilterme.

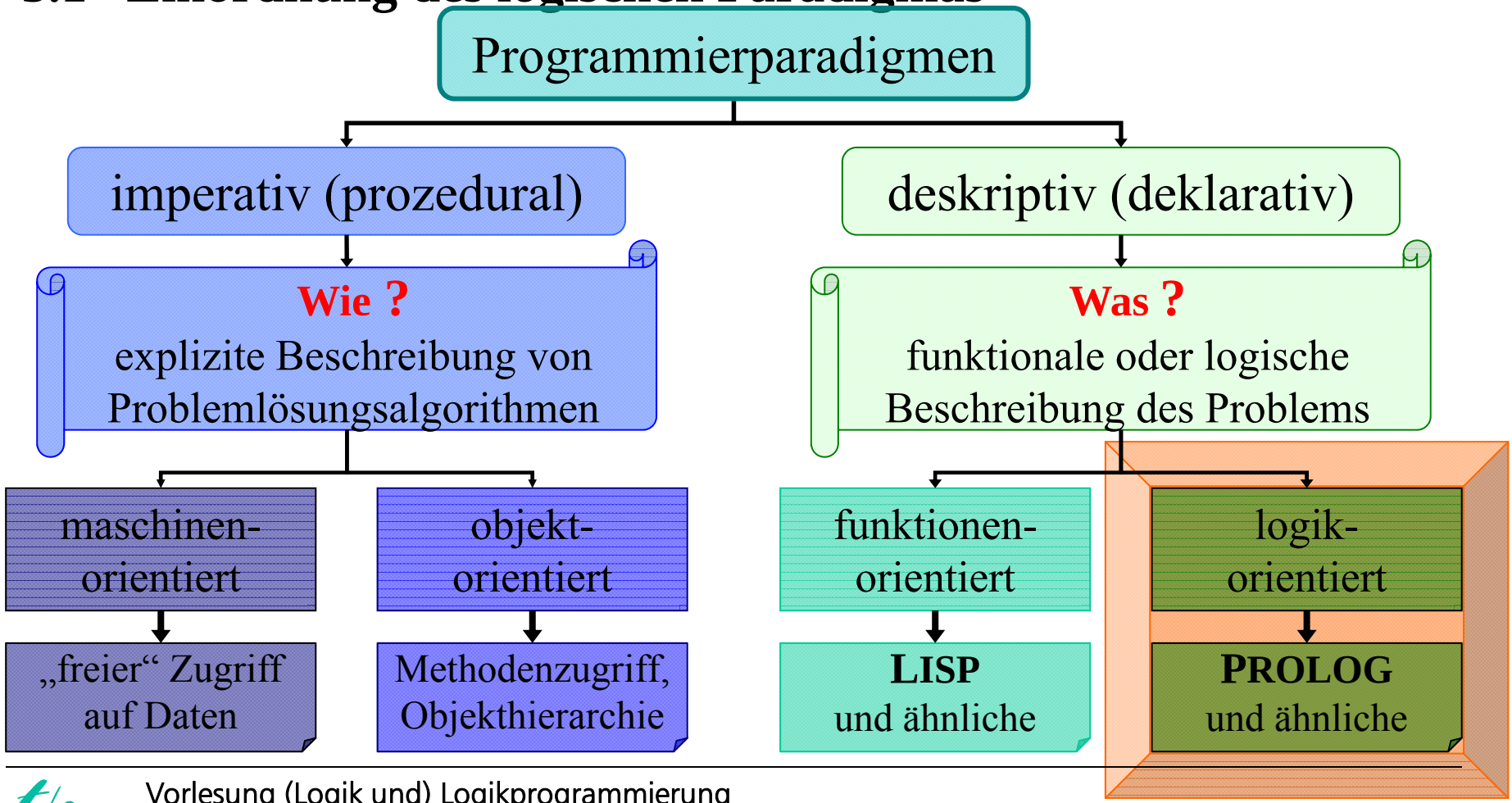
Algorithmus zur Bestimmung des allgemeinsten Unifikators 2er Terme

input: s , t **output**: Unifizierbarkeitsaussage, ggf. $\vartheta = m.g.u.(s, t)$

$i := 0$; $\vartheta_i := \varnothing$
$s_i := s$; $t_i := t$
▶ s_i und t_i identisch?
ja ⇒ **s und t sind unifizierbar**, $\vartheta := \vartheta_i = m.g.u.(s,t)$ *(fertig)*
nein ⇒ Bilde die Unterscheidungsterme s_i^* und t_i^*
s_i^* oder t_i^* Variable?
nein ⇒ **s und t sind nicht unifizierbar** *(fertig)*
ja ⇒ sei *(o.B.d.A.)* s_i^* eine Variable
$s_i^* \subseteq t_i^*$? *(enthält t_i^* die Variable s_i^* ?)*
ja ⇒ **s und t sind nicht unifizierbar** *(fertig)*
nein ⇒ $\vartheta' := \{[s,t']: [s,t] \in \vartheta, t' := t|_{s^* \rightarrow t^*}\} \cup \{[\, s_i^* , t_i^* \,]\}$
$s' := \vartheta'(s_i)$; $t' := \vartheta'(t_i)$; $i := i+1$;
$\vartheta_i := \vartheta'$; $s_i := s'$; $t_i := t'$; gehe zu ▶

3 Logische Programmierung

3.1 Einordnung des logischen Paradigmas



„deskriptives“ Programmierparadigma =

Problembeschreibung

- Die Aussagenmenge $M = \{K_1, \ldots, K_n\}$, über denen gefolgert wird, wird in Form von Fakten und Regeln im PK1 notiert.
- Eine mutmaßliche Folgerung (Hypothese) H wird in Form einer Frage als negierte Hypothese hinzugefügt.

\+ Programmverarbeitung

- Auf der Suche eines Beweises für $M \models H$ werden durch mustergesteuerte Prozedur-Aufrufe Resolutions-Schritte zusammengestellt.
- Dem „Programmierer“ werden (begrenzte) Möglichkeiten gegeben, die systematische Suche zu beeinflussen.

3.2 Syntax

Syntax von Klauseln

	Syntax	Beispiel
Fakt	***praedikatensymbol(term,...term) .***	**liefert(xy_ag,motor,vw).**
Regel	***praedikatensymbol(term,...term) :-*** ***praedikatensymbol(term,...term) ,*** ***... ,*** ***praedikatensymbol(term,...term) .***	**konkurrenten(Fa1,Fa2) :-** **liefert(Fa1,Produkt,_),** **liefert(Fa2,Produkt,_).**
Frage	***?- praedikatensymbol(term,...term) ,*** ***... ,*** ***praedikatensymbol(term,...term) .***	**?- konkurrenten(ibm,X),** **liefert(ibm,_,X).**

Syntax von Termen (1)

		Syntax	Beispiele
Konstante	Name	Zeichenfolge, beginnend mit Kleinbuchstaben, die Buchstaben, Ziffern und _ enthalten kann.	**otto_1 , tisch, hund**
		beliebige Zeichenfolge in "..." geschlossen	**"Otto", "r@ho"**
		Sonderzeichenfolge	**€%&§$€**
	Zahl	Ziffernfolge, ggf. mit Vorzeichen, Dezimalpunkt und Exponentendarstellung	**3, -5, 1001, 3.14E-12**
Variable	allg.	Zeichenfolge, mit Großbuchstaben oder _ beginnend	**X, Was, _alter**
	anonym	Unterstrich	**_**

Syntax von Termen (2)

		Syntax	Beispiele
struk-tu-rier-ter Term	allg.	***funktionssymbol(term , ... , term)***	**nachbar(chef(X))**
	Liste	leere Liste	**[ ]**
		[term | restliste]	**[mueller | [mayer | [ ]]]**
		[term , term , ... , term]	**[mueller, mayer, schulze]**

BACKUS-NAUR-Form (**Terminale**, Nichtterminale)

PROLOG-Programm	::= Wissensbasis Hypothese
Wissensbasis	::= Klausel | Klausel Wissensbasis
Klausel	::= Fakt | Regel
Fakt	::= Atomformel .
Atomformel	::= Prädikatensymbol (Termfolge)
Prädikatensymbol	::= Name
Name	::= Kleinbuchstabe | Kleinbuchstabe Restname | “ Zeichenfolge “ | Sonderzeichenfolge
RestName	::= Kleinbuchstabe | Ziffer | _ | Kleinbuchstabe RestName | Ziffer RestName | _ RestName

...(siehe Skript)

3.3 PROLOG aus logischer Sicht

Was muss der Programmierer tun?

- Formulierung einer Menge von Fakten und Regeln (kurz: Klauseln), d.h. einer Wissensbasis $M \equiv \{K_1,...,K_n\}$
- Formulierung einer negierten Hypothese (Frage, Ziel)

$$\neg H \equiv \neg \bigwedge_{i=1}^{m} H_i \equiv false \leftarrow \bigwedge_{i=1}^{m} H_i$$

Was darf der Programmierer erwarten?

- Dass das „Deduktionstool“ PROLOG ***M*** |= ***H*** zu zeigen versucht, d.h.

$$kt(\bigwedge_{i=1}^{n} K_i \wedge \neg H)$$

- ..., indem systematisch die Resolutionsmethode auf $\neg H$ und eine der Klauseln aus M angewandt wird, solange bis $\neg H \equiv false \leftarrow true$ entsteht

3.3.1 Formulierung von Wissensbasen (1)

BSP1.PRO

- Yoshihito und Sadako sind die Eltern von Hirohito.
- Kuniyoshi und Chikako sind die Eltern von Nagako.
- Akihito‘s und Hitachi‘s Eltern sind Hirohito und Nagako.
- Der Großvater ist der Vater des Vaters oder der Vater der Mutter.
- Geschwister haben den gleichen Vater und die gleiche Mutter.

```
vater_von(yoshihito,hirohito).
mutter_von(sadako,hirohito).
vater_von(kunioshi,nagako).
mutter_von(chikako,nagako).
vater_von(hirohito,akihito).
vater_von(hirohito,hitachi).
mutter_von(nagako,akihito).
mutter_von(nagako,hitachi).
grossvater_von(G,E) :-
  vater_von(G,V), vater_von(V,E).
grossvater_von(G,E) :-
  vater_von(G,M),mutter_von(M,E).
geschwister(X,Y) :-
  vater_von(V,X), vater_von(V,Y),
  mutter_von(M,X), mutter_von(M,Y).
```

Auf http://www.tu-ilmenau.de/ki/lehre/ kann man sich Visual Prolog herunterladen und alle Beispiele aus Vorlesung und Übung ausprobieren.
Visual Prolog© benötigt aber einen Deklarationsteil für Datentypen und Aritäten der Prädikate, der für die o.g. Wissensbasis so aussieht:

BSP1.PRO

```
domains
   person = symbol
predicates
   vater_von(person,person)          % bei nichtdeterministischen Prädikaten
   mutter_von(person,person).        % kann man „nondeterm“ vor das
   grossvater_von(person,person)     % Prädikat schreiben, um die
   geschwister(person,person)        % Kompilation effizienter zu machen
clauses
< Wissensbasis einfügen und Klauseln gleichen Kopfprädikates gruppieren>
goal
                  < Frage ohne „?-“ einfügen>
```

3.3.1 Formulierung von Wissensbasen (2)

BSP2.PRO

- Kollegen Meier und Müller arbeiten im Raum 1, Kollege Otto im Raum 2 und Kollege Kraus im Raum 3.
- Netzanschlüsse gibt es in den Räumen 2 und 3.
- Ein Kollege ist erreichbar, wenn er in einem Raum mit Netzanschluss arbeitet.
- 2 Kollegen können Daten austauschen, wenn sie im gleichen Raum arbeiten oder beide erreichbar sind.

```
• arbeitet_in(meier,raum_1).
  arbeitet_in(mueller,raum_1).
  arbeitet_in(otto,raum_2).
  arbeitet_in(kraus,raum_3).
• anschluss_in(raum_2).
  anschluss_in(raum_3).
• erreichbar(K) :-
      arbeitet_in(K,R),
      anschluss_in(R).
• koennen_daten_austauschen(K1,K2) :-
      arbeitet_in(K1,R),
      arbeitet_in(K2,R).
  koennen_daten_austauschen(K1,K2) :-
      erreichbar(K1),
      erreichbar(K2).
```

3.3.1 Formulierung von Wissensbasen (2)

Deklarationsteil

BSP2.PRO

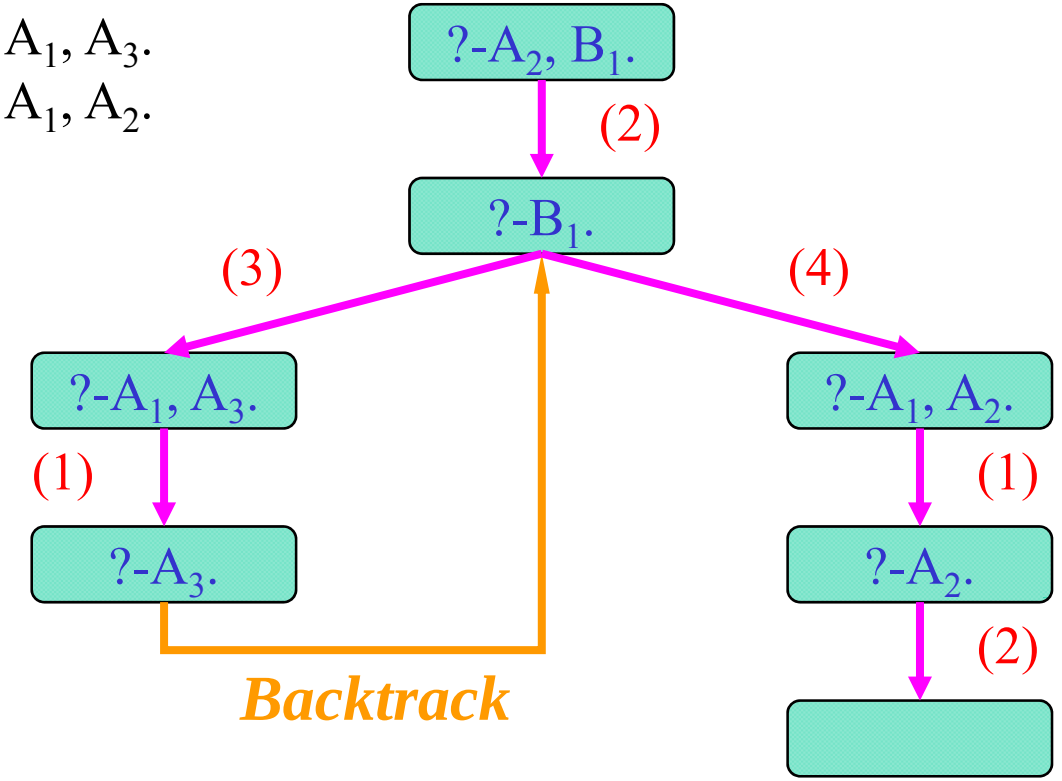
```
domains
        person, raum: symbol
predicates
        arbeitet_in(person,raum)
        ansluss_in(raum)
        erreichbar(person)
        koennen_daten_austauschen(person,person) :-
clauses
…
goal
…
```

Wissensbasis:
(1) A_1.
(2) A_2.
(3) B_1 :- A_1, A_3.
(4) B_1 :- A_1, A_2.

Ziel (Frage, Hypothese)**:** ?- A_2, B_1.

Veranschaulichung durch Suchbaum:

- Knoten = akt. Ziel
- Kante = Resolutionsschritt
- Markierung = Resolutionsklausel



Tiefensuche mit Backtrack

Es werden anwendbare Klauseln für das erste Teilziel gesucht. Gibt es ...

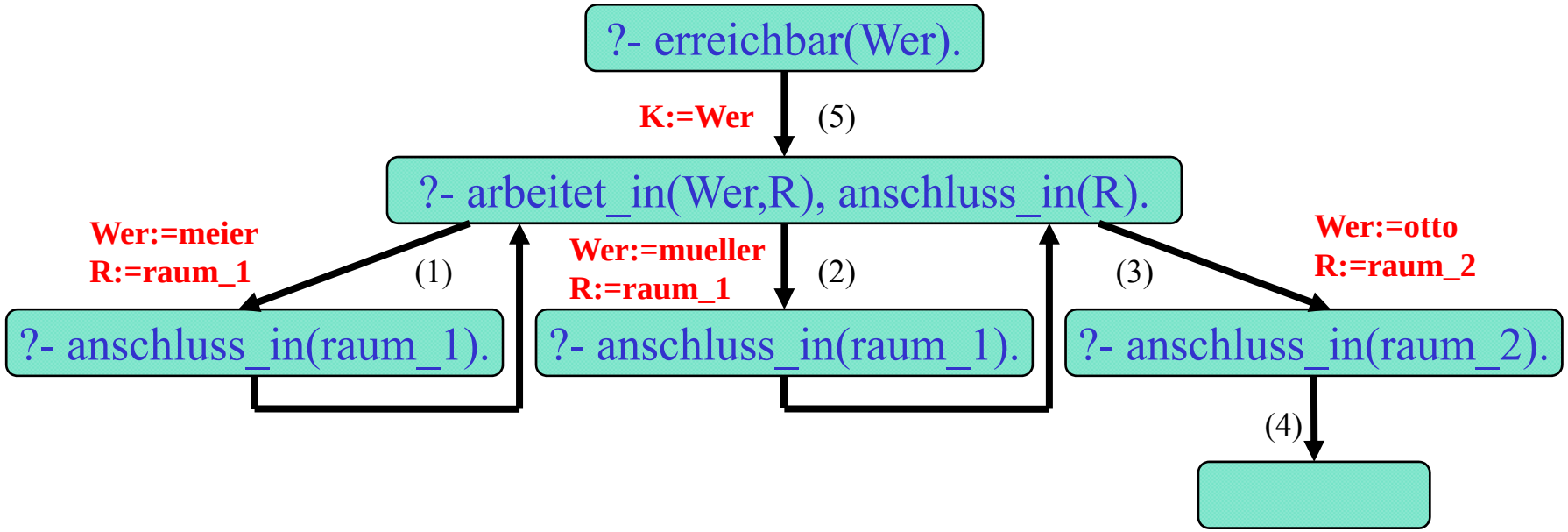
- ... genau eine, so wird das 1. Teilziel durch deren Körper ersetzt.
- ... mehrere, so wird das aktuelle Ziel inklusive alternativ anwendbarer Klauseln im Backtrack-Keller abgelegt und die am weitesten oben stehende Klausel angewandt.
- ... keine (mehr), so wird mit dem auf dem Backtrack-Keller liegendem Ziel die Bearbeitung fortgesetzt.

Dies geschieht solange, bis

- das aktuelle Ziel leer ist oder
- keine Klausel (mehr) anwendbar ist und der Backtrack-Keller leer ist.

(1) arbeitet_in(meier,raum_1).
(2) arbeitet_in(mueller,raum_1).
(3) arbeitet_in(otto,raum_2).
(4) anschluss_in(raum_2).
(5) erreichbar(K) :- arbeitet_in(K,R), anschluss_in(R).

Zusätzliche Markierung der Kanten mit der Variablenersetzung (dem **Unifikator**).



3.4 PROLOG aus prozeduraler Sicht

‘n Beispiel: die „Hackordnung“

```
(1)  chef_von(mueller,mayer).
(2)  chef_von(mayer,otto).
(3)  chef_von(otto,walter).
(4)  chef_von(walter,schulze).

(5)  weisungsrecht(X,Y) :-
         chef_von(X,Y).
(6)  weisungsrecht(X,Y) :-
         chef_von(X,Z),
         weisungsrecht(Z,Y).
```

Deklarative Interpretation

In einem Objektbereich

I= { mueller, mayer, schulze, ... }

bildet das **Prädikat**

weisungsrecht(X,Y)

[X,Y] auf wahr ab, gdw.

- das Prädikat *chef_von(X,Y)* das Paar [X,Y] auf wahr abbildet oder
- es ein $Z \in I$ gibt, so dass
  - das Prädikat *chef_von(X,Z)* das Paar [X,Z] auf wahr abbildet und
  - das Prädikat *weisungsrecht(Z,Y)* das Paar [Z,Y] auf wahr abbildet.

‘n Beispiel: die „Hackordnung“

```
(1) chef_von(mueller,mayer).
(2) chef_von(mayer,otto).
(3) chef_von(otto,walter).
(4) chef_von(walter,schulze).

(5) weisungsrecht(X,Y) :-
        chef_von(X,Y).
(6) weisungsrecht(X,Y) :-
        chef_von(X,Z),
        weisungsrecht(Z,Y).
```

Prozedurale Interpretation

Die **Prozedur**

weisungsrecht(X,Y)

wird abgearbeitet, indem

1. die Unterprozedur *chef_von(X,Y)* abgearbeitet wird.
   Im Erfolgsfall ist die Abarbeitung beendet; anderenfalls werden
2. die Unterprozeduren *chef_von(X,Z)* und *weisungsrecht(Z,Y)* abgearbeitet; indem systematisch Prozedurvarianten beider Unterprozeduren aufgerufen werden. Dies geschieht bis zum Erfolgsfall oder erfolgloser erschöpfender Suche.

deklarative Interpretation	**prozedurale Interpretation**
Prädikat	Prozedur
Ziel	Prozeduraufruf
Teilziel	Unterprozedur
Klauseln mit gleichem Kopfprädikat	Prozedur-varianten
Klauselkopf	Prozedurkopf
Klauselkörper	Prozedurrumpf

Die Gratwanderung zwischen Wünschenswertem und technisch Machbarem erfordert mitunter „Prozedurales Mitdenken“, um

1. eine gewünschte Reihenfolge konstruktiver Lösungen zu erzwingen,
2. nicht terminierende (aber – deklarativ, d.h. logisch interpretiert – völlig korrekte) Programme zu vermeiden,
3. seiteneffektbehaftete Prädikate sinnvoll einzusetzen,
4. (laufzeit-) effizienter zu programmieren und
5. das Suchverfahren gezielt zu manipulieren.

Programm inkl. Deklarationsteil BSP3.PRO

```
domains
        person = symbol

predicates
        chef_von(person,person)
        weisungsrecht(person,person)

clauses
chef_von(mueller,mayer).
chef_von(mayer,otto).
chef_von(otto,walter).
chef_von(walter,schulze).

weisungsrecht(X,Y) :-  chef_von(X,Y).
weisungsrecht(X,Y) :-  chef_von(X,Z), weisungsrecht(Z,Y).

goal
weisungsrecht(Wer,Wem).
```

Prädikate zur Steuerung der Suche nach einer Folge von Resolutionsschritten

! (cut)

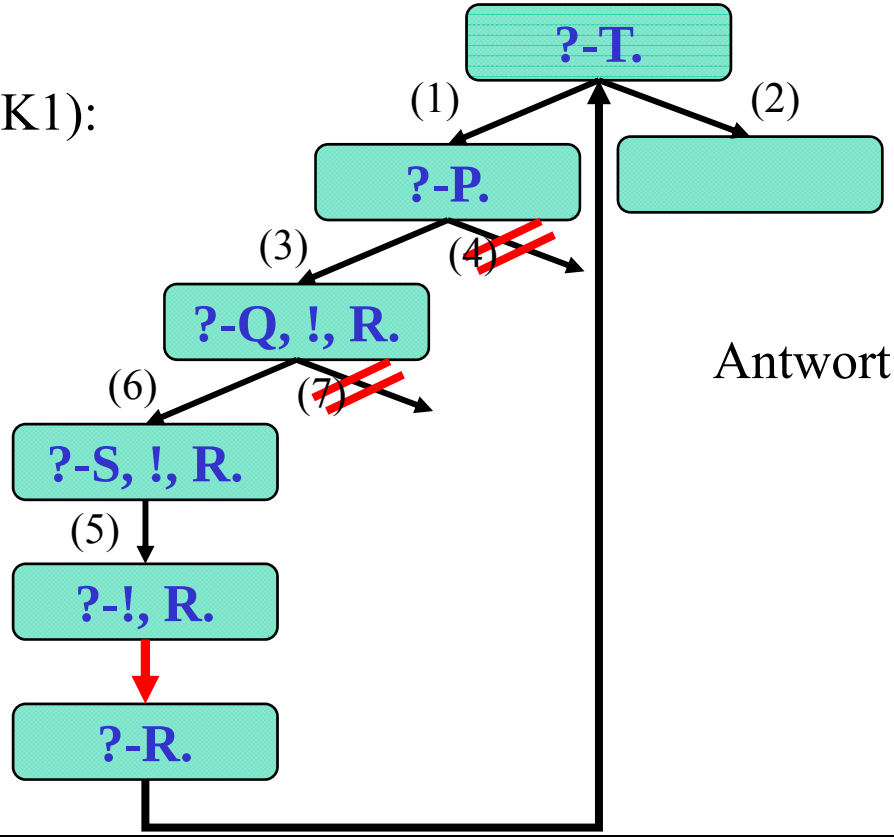
Das Prädikat !/0 ist stets wahr.
In Klauselkörpern eingefügt verhindert es ein Backtrack der hinter !/0 stehenden Teilziele zu den vor !/0 stehenden Teilzielen sowie zu alternativen Klauseln des gleichen Kopfprädikats.
Die Verarbeitung von !/0 schneidet demnach alle vor der Verarbeitung verbliebenen Lösungswege betreffenden Prozedur ab.

Prädikate zur Steuerung der Suche: ***!/0***

‘n Beispiel
(P, Q, R, S, T: Atomformeln des PK1):

(1) T :- P.
(2) T.
(3) P :- Q, !, R.
(4) P :- S.
(5) S.
(6) Q :- S.
(7) Q.

?- T.



Antwort: ***ja***

Prädikate zur Steuerung der Suche nach einer Folge von Resolutionsschritten

fail

Das Prädikat fail/0 ist stets falsch.
In Klauselkörpern eingefügt löst es ein Backtrack aus bzw. führt zum Misserfolg, falls der Backtrack-Keller leer ist, d.h. falls es keine verbleibenden Lösungswege (mehr) gibt.

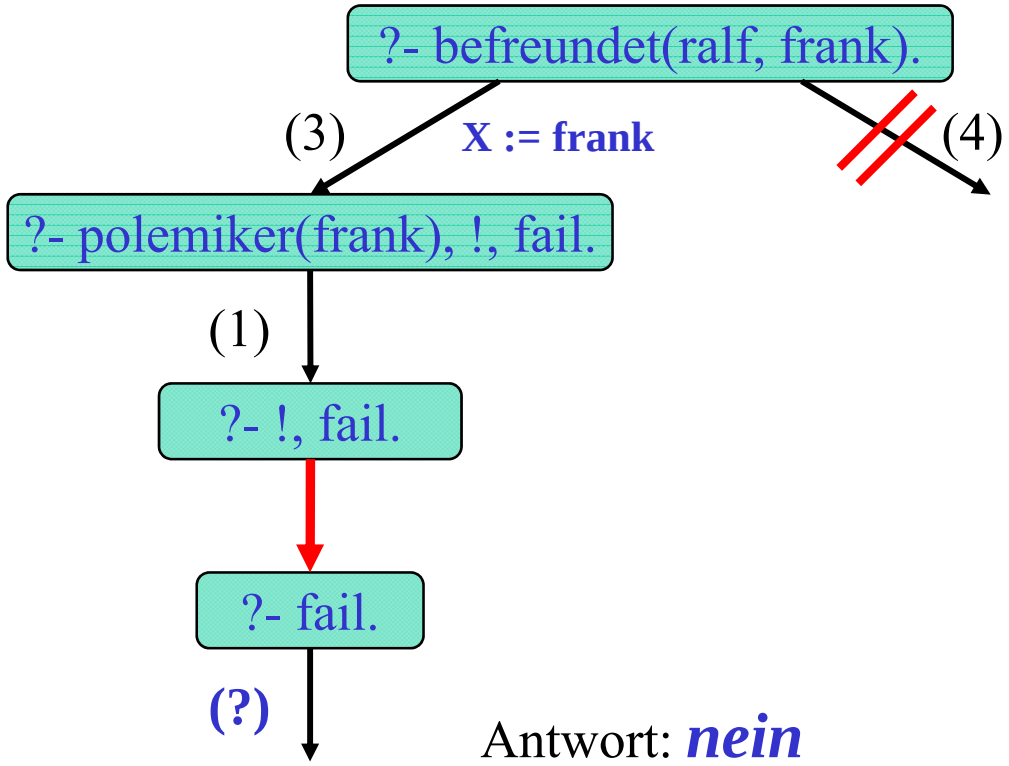
Prädikate zur Steuerung der Suche: ***fail/0***

BSP4.PRO

wieder ‘n Beispiel:

```
(1)  polemiker(frank).
(2)  polemiker(uwe).
(3)  befreundet(ralf, X) :-
         polemiker(X),
         !,
          fail.
(4)  befreundet(ralf, X).
```

?- befreundet(ralf, frank).



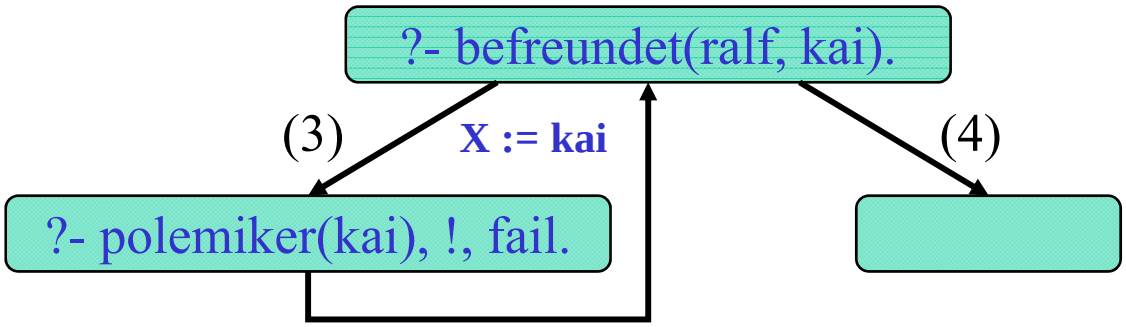
Antwort: ***nein***

Prädikate zur Steuerung der Suche: ***fail/0***

BSP4.PRO

gleiches Beispiel mit einer anderen Frage:

```
(1) polemiker(frank).
(2) polemiker(uwe).
(3) befreundet(ralf, X) :-
        polemiker(X),
        !,
        fail.
(4) befreundet(ralf, X).
```



Antwort: ***ja***

?- befreundet(ralf, kai).

3.5 Listen und rekursive Problemlösungsstrategien

Listen

1. **[ ]** ist eine Liste.
2. Wenn **T** ein Term und **L** eine Liste ist, dann ist
   (a) **[T | L]** eine Liste.
   (b) **T.L** eine Liste.
   (c) **. (T,L)** eine Liste.
   Das erste Element **T** heißt Listenkopf, **L** heißt Listenkörper oder Restliste.
3. Wenn t_1 , ... , t_n Terme sind, so ist **[**t_1 , ... , t_n **]** eine Liste.
4. Weitere Notationsformen von Listen gibt es nicht.

ungebräuchlich

Listen als kompakte Wissensrepräsentation: *ein bekanntes Beispiel*

```
arbeitet_in(meier, raum_1).
arbeitet_in(mueller, raum_1).

arbeitet_in(otto, raum_2).
arbeitet_in(kraus, raum_3).

anschluss_in(raum_2).
anschluss_in(raum_3).
```

```
arbeiten_in( [meier, mueller] , raum_1).

arbeiten_in( [otto] , raum_2 ).
arbeiten_in( [kraus] , raum_3 ).

anschluesse_in( [raum_2, raum_3] ).
```

Rekursion in der Logischen Programmierung

Eine Prozedur heißt (direkt) **rekursiv**, wenn in mindestens einem der Klauselkörper ihrer Klauseln ein erneuter Aufruf des Kopfprädikates erfolgt.

Ist der Selbstaufruf die letzte Atomformel des Klauselkörpers der letzten Klausel dieser Prozedur - bzw. wird er es durch vorheriges „Abschneiden“ nachfolgender Klauseln mit dem Prädikat !/0 - , so spricht man von **Rechtsrekursion**; anderenfalls von **Linksrekursion**.

Eine Prozedur heißt **indirekt rekursiv**, wenn bei der Abarbeitung ihres Aufrufes ein erneuter Aufruf derselben Prozedur erfolgt.

Wissensverarbeitung mit Listen: *das bekannte Beispiel*

```
erreichbar(K) :-   arbeitet_in(K,R),
                   anschluss_in(R).

koennen_daten_austauschen(K1,K2) :-
          arbeitet_in(K1,R),
          arbeitet_in(K2,R).

koennen_daten_austauschen(K1,K2) :-
          erreichbar(K1),
          erreichbar(K2).
```

```
erreichbar(K) :-   anschluesse_in(Rs),
                   member(R, Rs),
                   arbeiten_in(Ks, R),
                   member(K, Ks).
koennen_daten_austauschen(K1,K2) :-
          arbeiten_in(Ks,_),
          member(K1,Ks),
          member(K2,Ks).
koennen_daten_austauschen(K1,K2) :-
          erreichbar(K1),
          erreichbar(K2).
```

BSP5.PRO

```
domains
person, raum = symbol
raeume = raum*
personen = person*

predicates
arbeiten_in(personen, raum)
anschluesse_in(raeume)
erreichbar(person)
koennen_daten_austauschen(person,person)
member(person,personen)
member(raum,raeume)
```

```
clauses
… (siehe oben)

member(E,[E|_]).
member(E,[_|R]) :- member(E,R).

goal
erreichbar(Wer).
```

Unifikation 2er Listen

1. Zwei leere Listen sind (als identische Konstanten aufzufassen und daher) miteinander unifizierbar.
2. Zwei nichtleere Listen $[K_1|R_1]$ und $[K_2|R_2]$ sind miteinander unifizierbar, wenn ihre Köpfe (K_1 und K_2) und ihre Restlisten (R_1 und R_2) jeweils miteinander unifizierbar sind.
3. Eine Liste L und eine Variable X sind miteinander unifizierbar, wenn die Variable selbst nicht in der Liste enthalten ist.
   Die Variable X wird bei erfolgreicher Unifikation mit der Liste L instanziert: $X := L$.

Differenzlisten: *eine intuitive Erklärung*

Eine Differenzliste $L_1 - L_2$ besteht aus zwei Listen L_1 und L_2 und wird i.allg. als

$$[L_1,L_2]$$

oder (bei vorheriger Definition eines pre- bzw. infix notierten Funktionssymbols -/2) als $-(L_1,L_2)$ bzw. $L_1 - L_2$ notiert.
Sie wird (vom Programmierer, nicht vom PROLOG-System!) als eine Liste interpretiert, deren Elemente sich aus denen von L_1 abzüglich derer von L_2 ergeben.

Differenzlisten verwendet man typischerweise, wenn häufig Operationen am Ende von Listen vorzunehmen sind.

Differenzlisten: *Eine Definition*

1. **Die Differenz aus einer leeren Liste und einer (beliebigen) Liste ist die leere Liste:**
   [ ] – L = [ ]
2. **Die Differenz aus einer Liste *[E|R]* und der Liste *L* , welche *E* enthält, ist die Liste *D* , wenn die Differenz aus *R* und *L* (abzügl. E) die Liste *D* ist:**
   [E|R] – L = D , *wenn* E ∈ L *und* R – (L-[E]) = D
3. **Die Differenz aus einer Liste *[E|R]* und einer Liste *L* , welche *E* nicht enthält, ist die Liste *[E|D]* , wenn die Differenz aus *R* und *L* die Liste *D* ist: [E|R] – L = [E|D] , *wenn* E ∉ L *und* R – L = D**

Differenzlisten: *Ein Interpreter interpret (Differenzliste,Interpretation)* BSP6.PRO

```
(1) interpret( [ [ ] , _ ] , [ ] ) .
(2) interpret( [ [E|R] , L] , D ) :- loesche(E , L, L1), ! , interpret( [ R , L ] , D ) .
(3) interpret( [ [E|R] , L] , [E|D] ) :- interpret( [ R , L ] , D ) .
(4) loesche(E, [E|R], R) :-!.
(5) loesche(E, [K|R], [K|L]) :- loesche(E,R,L).
```

BSP6.PRO

Visual Prolog Programm

```
domains

element = symbol
liste = element*
listenliste = liste*

predicates

interpret(listenliste,liste)
loesche(element,liste,liste)

clauses

interpret( [ [ ] , _ ] , [ ] ) .
interpret( [ [E|R] , L] , D ) :-
        loesche(E , L, L1), ! ,
        interpret( [ R , L ] , D ) .
interpret( [ [E|R] , L] , [E|D] ) :-
        interpret( [ R , L ] , D ) .
loesche(E, [E|R], R) :-!.
loesche(E, [K|R], [K|L]) :-
        loesche(E,R,L).

goal

interpret([[a,b,c,d],[a,b]],X).
```

Differenzlisten: *Eine Anwendung* BSP7.PRO

```
quicksort(UnSortiert,Sortiert) :- qsort( UnSortiert , [ Sortiert, [ ] ] ) .
qsort( [ ] , [ Rest , Rest ] ) .
qsort( [ E | UnSortiert ] , [ Sortiert , Rest ] ) :-
      partition( UnSortiert , E , Kleinere , Groessere ) ,
      qsort( Groessere , [ Sortiert1 , Rest ] ) ,
      qsort( Kleinere , [ Sortiert , [ E | Sortiert1 ] ] ) .
partition( [ ] , _ , [ ] , [ ] ) .
partition( [ K | R ] , E , Kl , [ K | Gr ] ) :-
      K > E , ! , partition( R , E , Kl , Gr ) .
partition( [ K | R ] , E , [ K | Kl ] , Gr ) :-
      partition( R , E , Kl , Gr ) .
```

3.6 Prolog-Fallen

3.6.1 Nicht terminierende Programme

Ursache: „ungeschickt“ formulierte (direkte oder indirekte) Rekursion

Fall 1:

3.6.1.1 Alternierende Zielklauseln

BSP8.PRO

Ein aktuelles Ziel wiederholt sich und die Suche nach einer Folge von Resolutionsschritten endet nie:

```
(1)     liegt_auf(X,Y)   :- liegt_unter(Y,X).
(2)     liegt_unter(X,Y) :- liegt_auf(Y,X).

?- liegt_auf( skript, pult ).
```

logisch korrekte Antwort: ***nein*** tatsächliche Antwort: *keine*

... oder die Suche nach Resolutionsschritten endet mit einem Überlauf des Backtrack-Kellers: BSP8.PRO

```
(1)     liegt_auf(X,Y)    :- liegt_unter(Y,X).
(2)     liegt_auf( skript , pult ).
(3)     liegt_unter(X,Y)  :- liegt_auf(Y,X).

?- liegt_auf( skript , pult ).
```

logisch korrekte Antwort: ***ja***

tatsächliche Antwort: *keine*
(Visual Prolog: Fehlermeldung)

Dieses Beispiel zeigt, dass das Suchverfahren „Tiefensuche mit Backtrack“ die Vollständigkeit des Inferenzverfahrens zerstört.

Fall 2:

3.6.1.1 Expandierende Zielklauseln

Das erste Teilziel wird in jeden Resolutionsschritt durch mehrere neue Teilziele ersetzt; die Suche endet mit einem Speicherüberlauf: BSP9.PRO

```
(1)     liegt_auf( nootebook , pult ).
(2)     liegt_auf( skript , notebook ).
(3)     liegt_auf(X,Y)    :- liegt_auf(X,Z), liegt_auf(Z,Y).

?- liegt_auf( handy , skript ).
```

logisch korrekte Antwort: ***nein*** tatsächliche Antwort: *keine*

Auch dieses Beispiel lässt sich so erweitern, dass die Hypothese offensichtlich aus der Wissensbasis folgt, die Umsetzung der Resolutionsmethode aber die Vollständigkeit zerstört: BSP9.PRO

```
(1)     liegt_auf( nootebook , pult ).
(2)     liegt_auf( skript , notebook ).
(3)     liegt_auf(X,Y)    :- liegt_auf(X,Z), liegt_auf(Z,Y).
(4)     liegt_auf( handy , skript ).

?- liegt_auf( handy , pult ).
```

logisch korrekte Antwort: ***ja***

tatsächliche Antwort: *keine*
(Visual Prolog: Fehlermeldung)

Auch dieses Beispiel zeigt, dass das Suchverfahren „Tiefensuche mit Backtrack“ die Vollständigkeit des Inferenzverfahrens zerstört.

3.6.2 Metalogische Prädikate und konstruktive Lösungen

Das Prädikat **not/1** hat eine Aussage als Argument und ist somit eine Aussage über eine Aussage, also metalogisch.
I.allg. ist **not/1** vordefiniert, kann aber mit Hilfe von **call/1** definiert werden.
call/1 hat Erfolg, wenn sein Argument - als Ziel interpretiert - Erfolg hat.

Beispiel: BSP10.PRO

```
(1) fleissig(horst).
(2) fleissig(martin).
(3) faul(X) :- not( fleissig(X) ).
(4) not(X) :- call(X), !, fail.
(5) not( _ ).
```

?- faul(horst).
Antwort: nein

?- faul(alex).
Antwort: ja

?- faul(Wer).
Antwort: nein
(Visual Prolog: Fehlermeldung)

Wider-spruch

... und Beweis der Un-voll-ständig-keit durch Metalogik

4 Typische Problemklassen für die Anwendung der Logischen Programmierung

4.1 Rekursive Problemlösungsstrategien

Botschaft # 1

Man muss ein Problem nicht in allen Ebenen überblicken, um eine Lösungsverfahren zu programmieren.

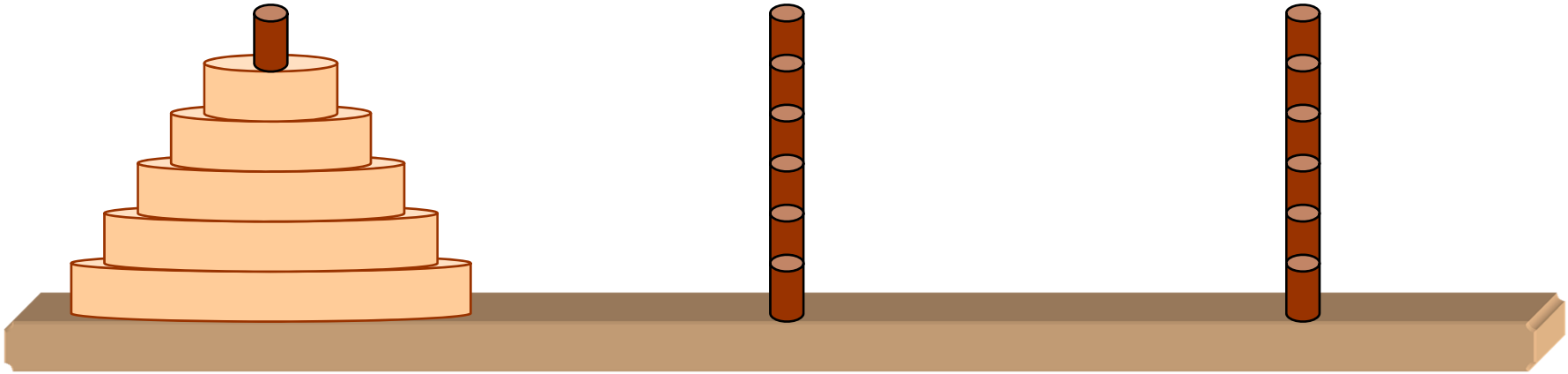
Es genügt die Einsicht,

1. ***wie man aus der Lösung eines einfacheren Problems die Lösung des präsenten Problems macht und***
2. ***wie es im Trivialfall zu lösen ist.***

Türme von Hanoi

Es sind N Scheiben von der linken Säule auf die mittlere Säule zu transportieren, wobei die rechte Säule als Zwischenablage genutzt wird.

Regeln:
1. Es darf jeweils nur eine Scheibe transportiert werden.
2. Die Scheiben müssen mit fallendem Durchmesser übereinander abgelegt werden.



(doppelt) rekursive Lösungsstrategie

- **N = 0** : Das Problem ist gelöst.
- **N > 0** :
  1. Man löse das Problem für **N-1** Scheiben, die von der Start-Säule zur Hilfs-Säule zu transportieren sind.
  2. Man lege eine Scheibe von der Start- zur Ziel-Säule.
  3. Man löse das Problem für **N-1** Scheiben, die von der Hilfs-Säule zur Ziel-Säule zu transportieren sind.

Prädikate

- hanoi(N)
  löst das Problem für **N** Scheiben
- verlege(N,Start,Ziel,Hilf)
  verlegt **N** Scheiben von **Start** nach **Ziel** unter Nutzung von **Hilf** als Ablage

Die Regeln zur Kodierung der Strategie BSP11.PRO

```
hanoi( N )  :- verlege( N , s1 , s2 , s3 ) .
verlege( 0 , _ , _ , _ ) .
verlege( N , S , Z , H ) :-
      N1 = N – 1 ,
      verlege( N1 , S , H , Z ) ,
      write("Scheibe von ", S," nach ", Z),
      verlege( N1 , H , Z , S ).
```

'ne Beispiel-Lösung

?- hanoi(4)

Scheibe von s1 nach s3
Scheibe von s1 nach s2
Scheibe von s3 nach s2
Scheibe von s1 nach s3
Scheibe von s2 nach s1
Scheibe von s2 nach s3
Scheibe von s1 nach s3
Scheibe von s1 nach s2
Scheibe von s3 nach s2
Scheibe von s3 nach s1
Scheibe von s2 nach s1
Scheibe von s3 nach s2
Scheibe von s1 nach s3
Scheibe von s1 nach s2
Scheibe von s3 nach s2

4.2 Sprachverarbeitung mit PROLOG

Botschaft # 2

Wann immer man Objekte mit Mustern vergleicht, z.B.

1. ***eine Struktur durch „Auflegen von Schablonen“ identifiziert,***
2. ***Gemeinsamkeiten mehrerer Objekte identifiziert, d.h. „eine Schablone entwirft“ oder***
3. ***„gemeinsame Beispiele für mehrere Schablonen“ sucht,***

mache man sich den Unifikations-Mechanismus zu nutzen.

Eine kontextfreie Grammatik (Chomsky-Typ 2) besteht aus

1. einem Alphabet **A**, welches die terminalen (satzbildenden) Symbole enthält
2. einer Menge nichtterminaler (satzbeschreibender) Symbole **N** (= Vokabular abzüglich des Alphabets: **N = V \ A)**
3. einer Menge von Ableitungsregeln **R ⊆ N×(N∪A)***
4. dem Satzsymbol **S ∈N**

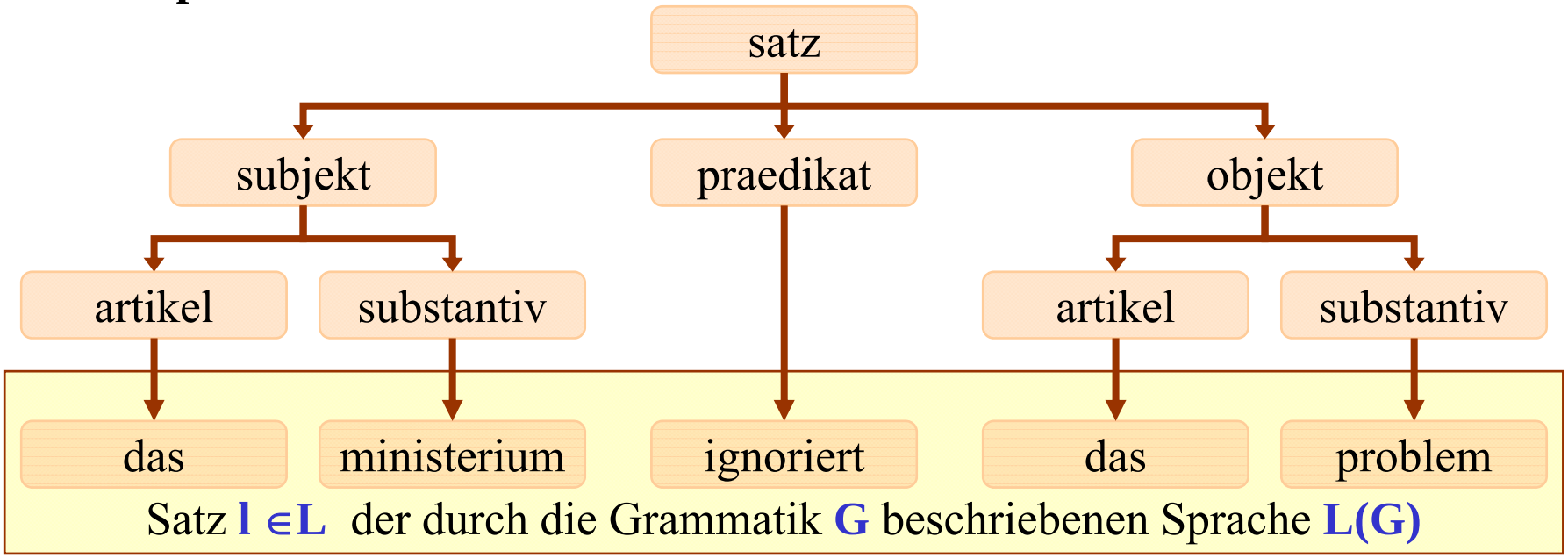
... und in PROLOG repräsentiert werden durch

1. 1-elementige Listen, welche zu satzbildenden Listen komponiert werden: [der] , [tisch] , [liegt] , ...
2. Namen, d.h. mit kleinem Buchstaben beginnende Zeichenfolgen: nebensatz , subjekt , attribut , ...
3. PROLOG-Regeln mit **l ∈N** im Kopf und **r ∈ (N∪A)*** im Körper
4. einen reservierten Namen: satz

Ableitungsbaum

Ein Ableitungsbaum beschreibt die grammatische Struktur eines Satzes. Seine Wurzel ist das Satzsymbol, seine Blätter in Hauptreihenfolge bilden den Satz.

ein Beispiel



Ein Beispiel

1. Alphabet

   ministerium , rektorat , problem , das , loest , ignoriert , verschaerft

2. nichtterminale Symbole

   satz , subjekt , substantiv , artikel , praedikat , objekt

3. Ableitungsregeln (in BACKUS-NAUR-Form)

satz	::=	subjekt praedikat objekt
subjekt	::=	artikel substantiv
objekt	::=	artikel substantiv
substantiv	::=	ministerium | rektorat | problem
artikel	::=	das
praedikat	::=	loest | ignoriert | verschaerft

4. Satzsymbol

   satz

Der Parser

BSP12.PRO

```
% satz ::= subjekt praedikat objekt
satz(L) :- subjekt(L1) ,
           praedikat(L2) ,
           objekt(L3) ,
           verkette( [L1,L2,L3] , L ).

% subjekt ::= artikel substantiv
subjekt(L) :- artikel(L1) ,
              substantiv(L2) ,
              verkette( [L1,L2] , L ).

% objekt ::= artikel substantiv
objekt(L) :- artikel(L1) ,
             substantiv(L2) ,
             verkette( [L1,L2] , L ).
```

```
%substantiv ::= ministerium | rektorat | problem
substantiv([ministerium]).
substantiv([rektorat]).
substantiv([problem]).

% artikel ::= das
artikel([das]).

%  praedikat ::= loest | ignoriert | verschaerft
praedikat([loest]).
praedikat([ignoriert]).
praedikat([verschaerft]).
```

Verketten einer Liste von Listen

BSP12.PRO

```
% die Liste ist leer
verkette( [ ] , [ ] ) .

% das erste Element ist eine leere Liste
verkette( [ [ ] | Rest ] , L ) :-
        verkette( Rest  , L ) .

% das erste Element ist eine nichtleere Liste
verkette([ [K | R ] | Rest ] , [ K | L ] ) :-
        verkette( [ R | Rest ] , L ) .
```

4.3 Die „Generate – and – Test“ Strategie

Botschaft # 3

Es ist mitunter leichter, für komplexe Probleme

1. ***eine potentielle Lösung zu „erraten“ und dazu***
2. ***ein Verfahren zu entwickeln, welches diese Lösung auf Korrektheit testet,***

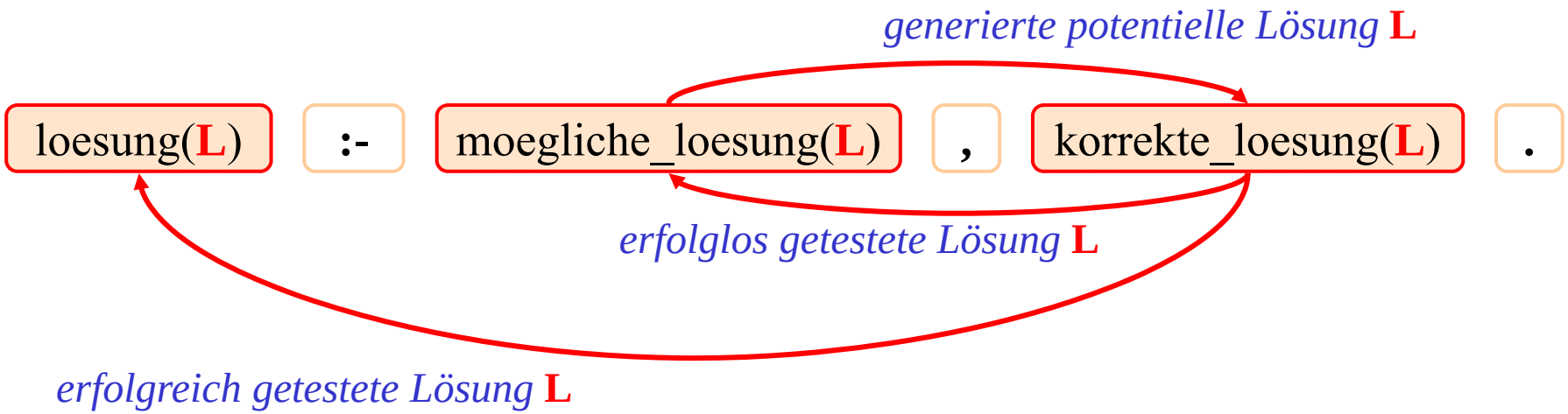
als zielgerichtet die korrekte Lösung zu entwerfen.
Hierbei kann man den Backtrack-Mechanismus nutzen.

Strategie

Ein Prädikat moegliche_loesung(L) generiert eine potentielle Lösung, welche von einem Prädikat korrekte_loesung(L) geprüft wird:

- Besteht **L** diesen Korrektheitstest, ist eine Lösung gefunden.
- Fällt **L** bei diesem Korrektheitstest durch, wird mit Backtrack das Prädikat moegliche_loesung(L) um eine alternative potentielle Lösung ersucht.

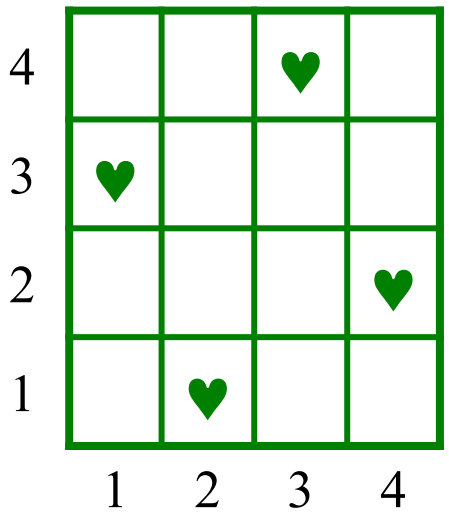
vgl.: Lösen NP-vollständiger Probleme, Entscheidung von Erfüllbarkeit



ein Beispiel

konfliktfreie Anordnung von N Damen auf einem $N \times N$ Schachbrett

Repräsentation der Anordnung als Term



- **eine Variante:** Liste strukturierter Terme
  [dame(Zeile,Spalte), ... , dame(Zeile,Spalte)]
  [dame(1,2), dame(2,4), dame(3,1), dame(4,3)]
- **noch ‘ne Variante:** Liste von Listen
  [[Zeile, Spalte] , ... , [Zeile,Spalte]]
  [[1,2] , [2,4] , [3,1] , [4,3]]
- **... und noch eine** (in die Wissensdarstellung etwas „natürliche“ Intelligenz investierende, den Problemraum enorm einschränkende) **Variante:**
  Liste der Spaltenindizes
  [Spalte_zu_Zeile_1, ..., Spalte_zu_Zeile_N]
  [2, 4, 1, 3]

BSP13.PRO

```
(1)  damen(N,L) :-
         moegliche_loesung(N,L),
         korrekte_loesung(L).
(2)  moegliche_loesung(N,L) :-
         erzeuge(N,L1),
         permutation(L1,L).
(3)  erzeuge( 0 , [ ]  ) :- ! .
(4)  erzeuge( N , [N|L] ) :-
         N1 = N – 1 ,
         erzeuge(N1, L).
(5)  permutation( [ ] , [ ] ).
(6)  permutation( [K|R] , L ) :-
           permutation( R , R1 ),
           fuege_ein(K , R1 , L ).
(7) fuege_ein( E , L , [E|L] ).
(8) fuege_ein( E , [K|R] , [K|R1]) :-
           fuege_ein( E , R , R1 ).
(9)  korrekte_loesung(L) :-
           teste(L , 1 ).
(10) teste( [E] , _ ).
(11) teste( [ E1 | [ E2 | R ] ] , N ) :-
           E2 = E1 + N, ! , fail .
(12) teste( [ E1 | [ E2 | R ] ] , N ) :-
           E2 = E1 – N , ! , fail .
(13) teste( [ E1 | [ E2 | R ] ] , N ) :-
           N1 = N + 1 ,
           teste( [E1 | R ] , N1 ),
           teste( [E2 | R ] , 1 ) .
```

4.4 Heuristische Problemlösungsmethoden

Botschaft # 4

Heuristiken sind

1. ***eine Chance, auch solche Probleme einer Lösung zuzuführen, für die man keinen (determinierten) Lösungsalgorithmus kennt und***
2. ***das klassische Einsatzgebiet zahlreicher KI-Tools – auch der Logischen Programmierung.***

Was ist eine Heuristik ?

Worin unterscheidet sich eine heuristische Problemlösungsmethode von einem Lösungsalgorithmus ?

Heuristiken bewerten die Erfolgsaussichten alternativer Problemlösungsschritte.

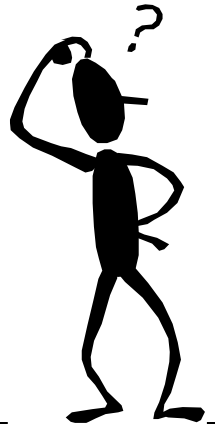
Eine solche Bewertung kann sich z.B. ausdrücken in

- einer quantitativen Abschätzung der „Entfernung“ zum gewünschten Ziel oder der „Kosten“ für das Erreichen des Ziels,
- einer quantitativen Abschätzung des Nutzens und/oder der Kosten der alternativen nächsten Schritte,
- eine Vorschrift zur Rangordnung der Anwendung alternativer Schritte, z.B. durch Prioritäten oder gemäß einer sequenziell abzuarbeitenden Checkliste.

Ein Beispiel:

Das Milchgeschäft meiner Großeltern in den 40er Jahren

- Der Milchhof liefert Milch in großen Kannen.
- Kunden können Milch nur in kleinen Mengen kaufen.
- Es gibt nur 2 Sorten geeichter Schöpfgefäße;
  sie fassen 0.75 Liter bzw. 1.25 Liter.
- Eine Kundin wünscht einen Liter Milch.



1. Wenn das große Gefäß leer ist, dann fülle es.
2. Wenn das kleine Gefäß voll ist, dann leere es.
3. Wenn beides nicht zutrifft, dann schütte so viel wie möglich vom großen in das kleine Gefäß.

Prädikat

miss_ab(VolGr , VolKl , Ziel , InhGr , InhKl)

Mit

- VolGr Volumen des großen Gefäßes
- VolKl Volumen des kleinen Gefäßes
- Ziel die abzumessende (Ziel-) Menge
- InhGr der aktuelle Inhalt im großen Gefäß
- InhKl der aktuelle Inhalt im kleinen Gefäß

Beispiel-Problem: **?- miss_ab(1.25 , 0.75 , 1 , 0 , 0)**

BSP14.PRO

```
% Abbruch , wenn Ziel erreicht
miss_ab( _ , _ , Z , _ , Z ) :-
  ! , write("Ziel erreicht !").
miss_ab( _ , _ , Z , Z , _ ) :-
  ! , write("Ziel erreicht!").

% Regel 1
miss_ab( VG, VK, Z, 0, IK ) :-
  ! , write("Großes Gefäß füllen !") ,
  miss_ab(VG , VK , Z , VG , IK).

% Regel 2
miss_ab( VG, VK, Z, IG, VK ) :-
  ! , write("Kleines Gefäß leeren !") ,
  miss_ab(VG , VK , Z , IG , 0).

% Regel 3
miss_ab( VG, VK, Z, IG, IK ) :-
   write("Schütte so viel wie möglich vom
          kleinen ins große Gefäß!")
   ! , schuette(VG, VK, IG, IK, NIG, NIK),
  miss_ab(VG, VK, Z, NIG, NIK).

% Fall 1 zu Regel 3: alles passt hinein
schuette( _, VK, IG, IK, 0, NIK) :-
  (VK – IK) >= IG , ! ,
  NIK = IK + IG.

% Fall 2 zu Regel 3: es bleibt ein Rest im
% kleinen Gefäß nach dem Schütten
schuette(VG, VK , IG, IK, NIG, VK) :-
  NIG = IG – (VK – IK).
```

4.5 Pfadsuche in gerichteten Graphen

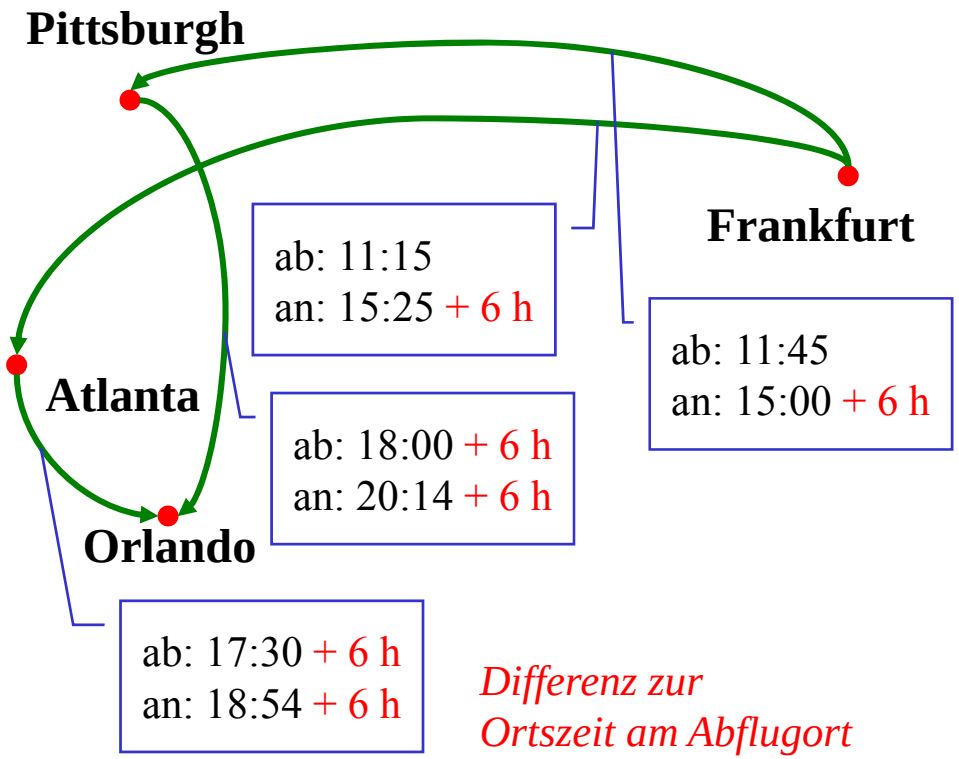
Botschaft # 5

1. ***Für die systematische Suche eines Pfades kann der Suchprozess einer Folge von Resolutionsschritten genutzt werden. Man muss den Suchprozess nicht selbst programmieren.***
2. ***Für eine heuristische Suche eines Pfades gilt Botschaft # 4: Sie ist das klassische Einsatzgebiet zahlreicher KI-Tools – auch der Logischen Programmierung.***

Anwendungen

- **Handlungsplanung** , z.B.
  - Suche einer Folge von Bearbeitungsschritten für ein Produkt, eine Dienstleistung, einen „Bürokratischen Vorgang“
  - Suche eines optimalen Transportweges in einem Netzwerk von Straßen-, Bahn-, Flugverbindungen
- **Programmsynthese** = Handlungsplanung mit ...
  - ... Schnittstellen für die Datenübergabe zwischen „Handlungsschritten“ (= Prozeduraufrufen) und
  - ... einem hierarchischen Prozedurkonzept, welches die Konfigurierung von „Programmbausteinen“ auf mehreren Hierarchie-Ebenen

Ein Beispiel: *Suche einer zeitoptimalen Flugverbindung*



Repräsentation als Faktenbasis

verbindung(Start,Zeit1,Ziel,Zeit2,Tag).

Start	Ort des Starts
Zeit1	Zeit des Starts
Ziel	Ort der Landung
Zeit2	Zeit der Landung
Tag	0, falls Zeit1 und Zeit2 am gleichen Tag und 1 ansonsten

verbindung(fra,z(11,45),ptb,z(21,0),0).
verbindung(fra,z(11,15),atl,z(21,25),0).
verbindung(ptb,z(24,0),orl,z(2,14),1).
verbindung(atl,z(23,30),orl,z(0,54),1).

In einer **dynamischen Wissensbasis** wird die bislang günstigste Verbindung in Form eines Faktes

guenstigste([v(Von,Zeit1,Nach,Zeit2,Tag), ...], Ankunftszeit, Tag).

festgehalten und mit den eingebauten Prädikaten assert(<Fakt>) - *zum Einfügen des Faktes* - und retract (<Fakt>) - *zum Entfernen des Faktes* - bei Bedarf aktualisiert.

Zum Beispiel

guenstigste([v(fra,z(11,45),ptb,z(21,00),0),v(ptb,z(24,0),orl,z(2,14),1)],z(2,14),1)

.

erklärt den Weg über Pittsburgh zum bislang günstigsten gefundenen Weg.

BSP15.PRO

Die Suchstrategie

```
start(S,Z,Zeit,Liste) :- suche(S,Z,Zeit,Liste), vergleiche(Liste), fail .
start(_,_,_,L) :-        guenstigste(L,_,_).
suche(S,Z,Zeit, [v(S,Z1,Z,Z2,T)] ) :- verbindung(S,Z1,Z,Z2,T), frueher_als(Zeit,0,Z1,T) ,!.
suche(S,Z,Zeit,[v(S,Z1,ZwZ,Z2,T)|R]) :- verbindung(S,Z1,ZwZ,Z2,T) ,
                                        frueher_als(Zeit,0,Z1,T) , suche(ZwZ,Z,Z2,R),
                                        not(member(v(S,_,_,_,_),R) ).
vergleiche(L) :- not(guenstigste(_,_,_)),!, ankunft(L,Zeit,T), assert(guenstigste(L,Zeit,T)).
vergleiche(L) :- ankunft(L,Zeit,T) , guenstigste( _ , Zeit1,T1),
                 frueher_als(Zeit,T,Zeit1,T1) , retract(guenstigste(_,_,_)) ,
                 assert(guenstigste(L,Zeit,T)) .
ankunft([v(_,_,_,Z2,T)] ,Z2,T) .
ankunft([_|R],Z,T) :- ankunft(R,Z,T) .
frueher_als(_,T1,_,T2) :-                    T2 > T1 , ! .
frueher_als(z(S1,_),T,z(S2,_),T) :-          S1 < S2  , ! .
frueher_als(z(S,Min1),T,z(S,Min2),T) :-      Min1 < Min2 .
```

4.6 „Logeleien“ als Prolog-Wissensbasen

Botschaft # 6

1. ***„Logeleien“ sind oft Aussagen über Belegungen von Variablen mit endlichem Wertebereich, ergänzt um eine Frage zu einem nicht explizit gegebenen Wert.***
2. ***Dabei handelt es sich um Grunde um eine Deduktionsaufgabe mit eine Hypothese zu einem mutmaßlichen Wert der gesuchten Variablen. Deshalb ist es oft auch mit dem „Deduktionstool“ Prolog lösbar, denn Prolog tut im Grunde nichts anderes als ein ziel-gerichtetes „Durchprobieren“ legitimer Deduktionsschritte im „Generate – and – Test“ – Verfahren.***

4.6.1 „Logeleien“ als Prolog-Wissensbasen - Das „Zebra-Rätsel“ BSP16.PRO

1. *Es gibt fünf Häuser.*
2. *Der Engländer wohnt im roten Haus.*
3. *Der Spanier hat einen Hund.*
4. *Kaffee wird im grünen Haus getrunken.*
5. *Der Ukrainer trinkt Tee.*
6. *Das grüne Haus ist (vom Betrachter aus gesehen) direkt rechts vom weißen Haus.*
7. *Der Raucher von Atem-Gold-Zigaretten hält Schnecken als Haustiere.*
8. *Die Zigaretten der Marke Kools werden im gelben Haus geraucht.*
9. *Milch wird im mittleren Haus getrunken.*
10. *Der Norweger wohnt im ersten Haus.*
11. *Der Mann, der Chesterfields raucht, wohnt neben dem Mann mit dem Fuchs.*
12. *Die Marke Kools wird geraucht im Haus neben dem Haus mit dem Pferd.*
13. *Der Lucky-Strike-Raucher trinkt am liebsten Orangensaft.*
14. *Der Japaner raucht Zigaretten der Marke Parliament.*
15. *Der Norweger wohnt neben dem blauen Haus.*

Jedes Haus ist in einer anderen Farbe gestrichen und jeder Bewohner hat eine andere Nationalität, besitzt ein anderes Haustier, trinkt ein von den anderen Bewohnern verschiedenes Getränk und raucht eine von den anderen Bewohnern verschiedene Zigarettensorte.

Fragen: (1) Wer trinkt Wasser? (2) Wem gehört das Zebra?

```
loesung(WT, ZB) :-
        haeuser(H), nationen(N), getraenke(G), tiere(T), zigaretten(Z),
        aussagentest(H,N,G,T,Z), wassertrinker(N,G,WT), zebrabesitzer(N,T,ZB).

tiere(X) :- permutation([fuchs, hund, schnecke, pferd, zebra],X).

nationen([norweger|R]) :-
        permmutation([englaender, spanier, ukrainer, japaner], R).      % erfüllt damit Aussage 10

getraenke(X) :- permutation([kaffee, tee, milch, osaft, wasser], X), a9(X). % erfüllt damit Aussage 9

zigaretten(X) :- permmutation([atemgold,kools, chesterfield, luckystrike, parliament], X).

haeuser(X) :- permutation([rot, gruen, weiss, gelb, blau], X), a6(X).

% parmutation/2, fuege_ein/3: siehe BSP13-PRO

wassertrinker([WT| _ ], [wasser| _ ], WT ) :- !.
wassertrinker([ _ |R1], [ _ |R2], WT) :- wassertrinker(R1, R2, WT).

zebrabesitzer([ZB| _ ], [zebra| _ ], ZB) :- !.
zebrabesitzer([ _ |R1], [ _ |R2], ZB) :- zebrabesitzer(R1, R2, ZB).

aussagentest(H,N,G,T,Z) :-
        a2(H,N), a3(N,T), a4(H,G), a5(N,G), a7(Z,T), a8(H,Z), a11(Z,T), a12(Z,T),
        a13(Z,G), a14(N,Z), a15(H,N).
```

BSP16.PRO

```
a2([rot| _ ], [englaender| _ ]) :- !.
a2([ _ |R1], [ _ |R2]) :- a2(R1, R2).

a3([spanier| _ ], [hund| _ ]) :- !.
a3([ _ |R1], [ _ |R2]) :- a3(R1, R2).

a4([gruen| _ ], [kaffee| _ ]) :- !.
a4([ _ |R1], [ _ |R2]) :- a4(R1, R2).

a5([ukrainer| _ ], [tee| _ ]) :- !.
a5([ _ |R1], [ _ |R2]) :- a5(R1, R2).

a6([weiss, gruen| _ ]) :- !.
a6([weiss| _ ]) :- !, fail.
a6([ _ |R]) :- a6(R).

a7([atemgold| _ ], [schnecke| _ ]) :- !.
a7([ _ |R1], [ _ |R2]) :- a7(R1, R2).

a8([gelb| _ ], [kools| _ ]) :- !.
a8([ _ |R1], [ _ |R2]) :- a8(R1, R2).

a9([ _ , _ , milch, _ , _ ]).

a11([chesterfield| _ ], [ _ ,fuchs| _ ]) :-!.
a11([ _ , chesterfield| _ ], [fuchs| _ ]) :- !.
a11([ _ |R1], [ _ |R2]) :- a11(R1, R2).

a12([kools| _ ], [ _ , pferd| _ ]) :- !.
a12([ _ , kools| _ ], [pferd| _ ] ) :- !.
a12([ _ |R1], [ _ |R2]) :- a12(R1, R2).

a13([luckystrike| _ ], [osaft| _ ]) :- !.
a13([ _ |R1], [ _ |R2]) :- a13(R1, R2).

a14([japaner| _ ], [parliament| _ ]) :- !.
a14([ _ |R1], [ _ |R2]) :- a14(R1, R2).

a15([blau| _ ], [ _, norweger| _ ]) :- !.
a15([ _ , blau| _ ], [norweger| _ ]) :- !.
a15([ _ |R1], [ _ |R2]) :- a15(R1, R2).
```

?- loesung(Wassertrinker, Zebrabesitzer).

Repräsentation eines Sudoku:

- Liste 9-elementliger Listen, die (von links nach rechts) die Zeilen (von oben nach unten) repräsentieren
- Elemente einer jeden eine Zeile repräsentierenden Liste:
  - Ziffer, falls dort im gegebenen Sudoku eine Ziffer steht
  - Anonyme Variable andernfalls

Beispiel:

	1			**3**	**4**			
				7			**2**	
3	**4**	**2**						
4	**3**	**8**						
						5	**6**	
				4	**9**			
8			**7**					**9**
			9		**5**	**1**		**7**
9		**3**						

Repräsentation:

```
L = [ [ _, 1, _, _, 3, 4, _, _, _ ],
      [ _, _, _, _, 7, _, _, 2, _ ],
      [ 3, 4, 2, _, _, _, _, _, _ ],
      [ 4, 3, 8, _, _, _, _, _, _ ],
      [ _, _, _, _, _, _, 5, 6, _ ],
      [ _, _, _, _, 4, 9, _, _, _ ],
      [ 8, _, _, 7, _, _, _, _, 9 ],
      [ _, _, _, 9, _, 5, 1, _, 7 ],
      [ 9, _, 3, _, _, _, _, _, _ ] ]
```

4.6.2 „Logeleien“ als Prolog-Wissensbasen - SUDOKU 数独 (すうどく) BSP17.PRO

```
sudoku(X,L) :- generiere_loesung(X,L), teste_loesung(L).

generiere_loesung([X1,X2,X3,X4,X5,X6,X7,X8,X9], [L1,L2,L3,L4,L5,L6,L7,L8,L9]):-
        perm(X1,L1),perm(X2,L2),perm(X3,L3),
        perm(X4,L4),perm(X5,L5),perm(X6,L6),
        perm(X7,L7),perm(X8,L8),perm(X9,L9).

% perm/2 bekommt Liste mit Ziffern und anonymen Variablen, liefert permutierte Ziffernliste ohne Variablen
perm(L,L1):- fehlende_ziffern(L,Ziffernliste), permutation(Ziffernliste,Einzufuegende_ziffern),
             einfuegen(L,Einzufuegende_ziffern,L1).

% fehlende_ziffern/2 bekommt Liste mit Ziffern und anonymen Variablen, ermittelt die fehlenden Ziffern
fehlende_ziffern(L,L1):- loesche_variablen(L,L2), pruefe([1,2,3,4,5,6,7,8,9],L2,[ ],L1).

% pruefe/4 bekommt die Liste [1,2,3,4,5,6,7,8,9], die Ergebnisliste L2 von loesche_variablen und einen
Akkumulator von Zahlen, die in der entsprechende Zeile noch fehlen (initial leer) und liefert diese zurück
pruefe([ ],_,A,A).
pruefe([K|R],L,A,L1) :- member(K,L), !, pruefe(R,L,A,L1).
pruefe([K|R],L,A,L1) :- pruefe(R,L,[K|A],L1).
```

```
member(E,[E|_]) :- !.
member(E,[_|R]) :- member(E,R).

loesche_variablen([ ],[ ]).
loesche_variablen([K|R],R1) :- free(K), !, loesche_variablen(R,R1).  % free/1 prüft Variableneigenschaft
loesche_variablen([K|R],[K|R1]) :- loesche_variablen(R,R1).

permutation([ ],[ ]).
permutation([K|R],L) :- permutation(R,L1), fuege_ein(K,L1,L).

fuege_ein(E,L,[E|L]).
fuege_ein(E,[K|R],[K|R1]) :- fuege_ein(E,R,R1).

% einfuegen/3 bekommt zwei Listen (gegebene Liste und fehlende Ziffern) und liefert eine Ziffernliste, in der
die fehlenden Ziffern in der permutierten Reihenfolge an den Stellen eingefügt sind, an denen Variablen stehen
einfuegen(L,[ ],L):-!.
einfuegen([K|R],NL,[K|R1]):-bound(K),!,einfuegen(R,NL,R1).
einfuegen([_|R],[N1|RN],[N1|R1]):-einfuegen(R,RN,R1).
```

4.6.2 „Logeleien“ als Prolog-Wissensbasen - SUDOKU 数独 (すうどく) BSP17.PRO

```
teste_loesung([L1,L2,L3,L4,L5,L6,L7,L8,L9]):- % zeilentest wg. Zeilengenerierungsmethode unnötig
        % teste_zeile(L1), teste_zeile(L2), teste_zeile(L3), teste_zeile(L4), teste_zeile(L5),
        % teste_zeile(L6), teste_zeile(L7), teste_zeile(L8), teste_zeile(L9),
        teste_spalten(L1,L2,L3,L4,L5,L6,L7,L8,L9),
        teste_block(L1,L2,L3),test_block(L4,L5,L6),test_block(L7,L8,L9).

% teste_zeile([ ]).
% teste_zeile(L):-verschieden(L).

teste_spalten([ ], [ ], [ ], [ ], [ ], [ ], [ ], [ ], [ ]).
teste_spalten([K1|R1],[K2|R2],[K3|R3],[K4|R4],[K5|R5],[K6|R6],[K7|R7],[K8|R8], [K9|R9]) :-
      verschieden([K1,K2,K3,K4,K5,K6,K7,K8,K9]), teste_spalten(R1,R2,R3,R4,R5,R6,R7,R8,R9).
teste_block([A1,A2,A3,A4,A5,A6,A7,A8,A9], [B1,B2,B3,B4,B5,B6,B7,B8,B9],
          [C1,C2,C3,C4,C5,C6,C7,C8,C9]):- verschieden([A1,A2,A3,B1,B2,B3,C1,C2,C3]),
                                          verschieden([A4,A5,A6,B4,B5,B6,C4,C5,C6]),
                                          verschieden([A7,A8,A9,B7,B8,B9,C7,C8,C9]).

verschieden([ ]).
verschieden([K|R]):- member(K,R), !, fail.
verschieden([_|R]):- verschieden(R).
```

4.6 Tools für die formale Logik

Botschaft # 7

Auch in der formalen Logik gibt es Deduktionsaufgaben, bei der Variablenbelegungen gesucht sind, welche eine Aussage wahr machen:

1. ***Meist geschieht das durch systematische Auswertung der Aussage, wozu das Suchverfahren von Prolog genutzt werden kann.***
2. ***Auch hier geht es oft um gesuchte Werte für Variablen. Deshalb ist es oft auch mit dem „Deduktionstool“ Prolog lösbar, denn Prolog tut im Grunde nichts anderes als ein ziel-gerichtetes „Durchprobieren“ legitimer Deduktionsschritte im „Generate – and – Test“ – Verfahren.***

Repräsentation von Aussagen als PROLOG-Term:

- true, false: *atom(true), atom(false)*
- A1 $\wedge$ A2: *und(A1,A2)*
- A1 $\vee$ A2: *oder(A1,A2)*
- $\neg$A: *nicht(A)*
- A1 $\rightarrow$ A2: *wenndann(A1,A2)*
- A1 $\leftarrow$A2: *dannwenn(A1,A2)*
- A1 $\leftrightarrow$ A2: *gdw(A1,A2)*

Erfüllbarkeitstest:

?- erfuellbar(gdw(wenndann(nicht(oder(atom(false),atom(X))),atom(Y)), atom(Z))).

X=true, Y=_, Z=true ; steht für 2 Modelle (eines mit Y = true und eines mit Y = false

X=false, Y=true, Z=true

```
erfuellbar(atom(true)).

erfuellbar(und(X,Y)) :- erfuellbar(X), erfuellbar(Y).

erfuellbar(oder(X,_)) :- erfuellbar(X).
erfuellbar(oder(_,X)) :- erfuellbar(X).

erfuellbar(nicht(atom(false))).                          % Implementierung mit
erfuellbar(nicht(und(X,_))) :- erfuellbar(nicht(X)).     % „!, fail“ geht nicht, da
erfuellbar(nicht(und(_,X))) :- erfuellbar(nicht(X)).     % diese keine konstruktiven
erfuellbar(nicht(oder(X,Y))) :- erfuellbar(nicht(X)), erfuellbar(nicht(Y)). % Lösungen liefern
erfuellbar(nicht(nicht(X))) :- erfuellbar(X).                               % würde.

erfuellbar(wenndann(X,_)) :- erfuellbar(nicht(X)).
erfuellbar(wenndann(X,Y)) :- erfuellbar(X), erfuellbar(Y).

erfuellbar(dannwenn(X,Y)) :- erfuellbar(wenndann(Y,X)).

erfuellbar(gdw(X,Y)) :- erfuellbar(X), erfuellbar(Y).
erfuellbar(gdw(X,Y)) :- erfuellbar(nicht(X)), erfuellbar(nicht(Y)).

% Für welche Belegungen von X, Y, und Z wird (¬(flase ∨ X))→Y)↔Z erfüllt?
?- erfuellbar(gdw(wenndann(nicht(oder(false,atom(X))),atom(Y)), atom(Z))).
```

Ketten von Konjunktionen und Disjunktionen als PROLOG-Listen:

- A1 ∧ A2 ∧ … ∧ An : *undverkettung([A1,A2, …, An])*
- A1 ∨ A2 ∨ … ∨ An : *oderverkettung([A1,A2, …, An])*

Erfüllbarkeitstest:

?- erfuellbar(undverkettung([true,X,Y,true]))
X = true Y = true
?- erfuellbar(oderverkettung([false,X,Y,false]))
X = true Y = _
X = _ Y = true

Implementierung:

```
erfuellbar(undverkettung([])).
erfuellbar(undverkettung([K|R])) :- erfuellbar(K),erfuellbar(undverkettung(R)).
erfuellbar(oderverkettung([K|_])) :- erfuellbar(K).
erfuellbar(oderverkettung([_|R])) :- erfuellbar(oderverkettung(R)).
```

TermauswerterFuerMehrwertigeLogiken.PRO

Repräsentation von Termen als PROLOG-Term:

- Wert: *atom(<reelle Zahl>)*
- A1 ∧ A2: *und(A1,A2)*
- A1 ∨ A2: *oder(A1,A2)*
- ¬A: *nicht(A)*
- A1 → A2: *wenndann(A1,A2)*
- A1 ←A2: *dannwenn(A1,A2)*
- A1 ↔ A2: *gdw(A1,A2)*
- A1 ∧ A2 ∧ … ∧ An : *undverkettung([A1,A2, …, An])*
- A1 ∨ A2 ∨ … ∨ An : *oderverkettung([A1,A2, …, An])*

Termauswertung:

?- hat_wert(und(atom(0.5),oder(atom(0.7),atom(0.3))),X).

X=0.5

TermauswerterFuerMehrwertigeLogiken.PRO

```
hat_wert(atom(X),X).
hat_wert(und(X,Y),Z) :- hat_wert(X,Z1), hat_wert(Y,Z2),min(Z1,Z2,Z).
hat_wert(oder(X,Y),Z) :- hat_wert(X,Z1), hat_wert(Y,Z2),max(Z1,Z2,Z).
hat_wert(nicht(X),Z) :- hat_wert(X,Z1), Z = 1-Z1.
hat_wert(wenndann(X,Y),Z) :- hat_wert(X,Z1), hat_wert(Y,Z2), Z3 = 1-Z1, max(Z2,Z3,Z).
hat_wert(dannwenn(X,Y),Z) :- hat_wert(wenndann(Y,X),Z).
hat_wert(undverkettung(L),W) :- listenmin(L,W).
hat_wert(oderverkettung(L),W) :- listenmax(L,W).
min(X,Y,X) :- X <= Y.          min(X,Y,Y) :- X > Y.
max(X,Y,X) :- X >= Y.          max(X,Y,Y) :- X < Y.
listenmin([K|R],W) :- hat_wert(K,W1), minimal(R,W1,W).
minimal([ ],A,A).
minimal([K|R],A,W) :- hat_wert(K,W1), W1 <= A, !, minimal(R,W1,W).
minimal([_|R],A,W) :- minimal(R,A,W).
listenmax([K|R],W) :- hat_wert(K,W1), maximal(R,W1,W).
maximal([ ],A,A).
maximal([K|R],A,W) :- hat_wert(K,W1), W1 >= A, !, maximal(R,W1,W).
maximal([_|R],A,W) :- maximal(R,A,W).

?-hat_wert(und(atom(0.5),oder(atom(0.7),atom(0.3))),X).
X = 0.5
```

Repräsentation von Termen als PROLOG-Term:

… siehe vorheriges Programm

Termauswertung unter Vorgabe des Wertebereiches:

*?- hat_wert(**[0,0.3, 0.5, 0.7, 1]** , und(atom(X),oder(atom(0.5),atom(0.7))),Wert).*
X=0, Wert=0
X=0.3, Wert=0.3
X=0.5, Wert=0.5
X=0.7, Wert=0.7
X=1, Wert=0.7

TermauswerterMitVariablen.PRO

```
hat_wert(WB,atom(X),X) :- member(X,WB).
hat_wert(WB,und(X,Y),Z) :- hat_wert(WB,X,Z1), hat_wert(WB,Y,Z2),
                           minrel(WB,Z1,Z2,Z).
hat_wert(WB,oder(X,Y),Z) :- hat_wert(WB,X,Z1), hat_wert(WB,Y,Z2),
                            maxrel(WB,Z1,Z2,Z).
hat_wert(WB,nicht(X),Z) :- hat_wert(WB,X,Z1), Z = 1-Z1.
hat_wert(WB,wenndann(X,Y),Z) :- hat_wert(WB,nicht(X),Z1), hat_wert(WB,Y,Z2),
                                maxrel(WB,Z1,Z2,Z).
hat_wert(WB,dannwenn(X,Y),Z) :- hat_wert(WB,wenndann(Y,X),Z).
hat_wert(WB,undverkettung(L),W) :- listenminrel(WB,L,W).
hat_wert(WB,oderverkettung(L),W) :- listenmaxrel(WB,L,W).
```

TermauswerterMitVariablen.PRO

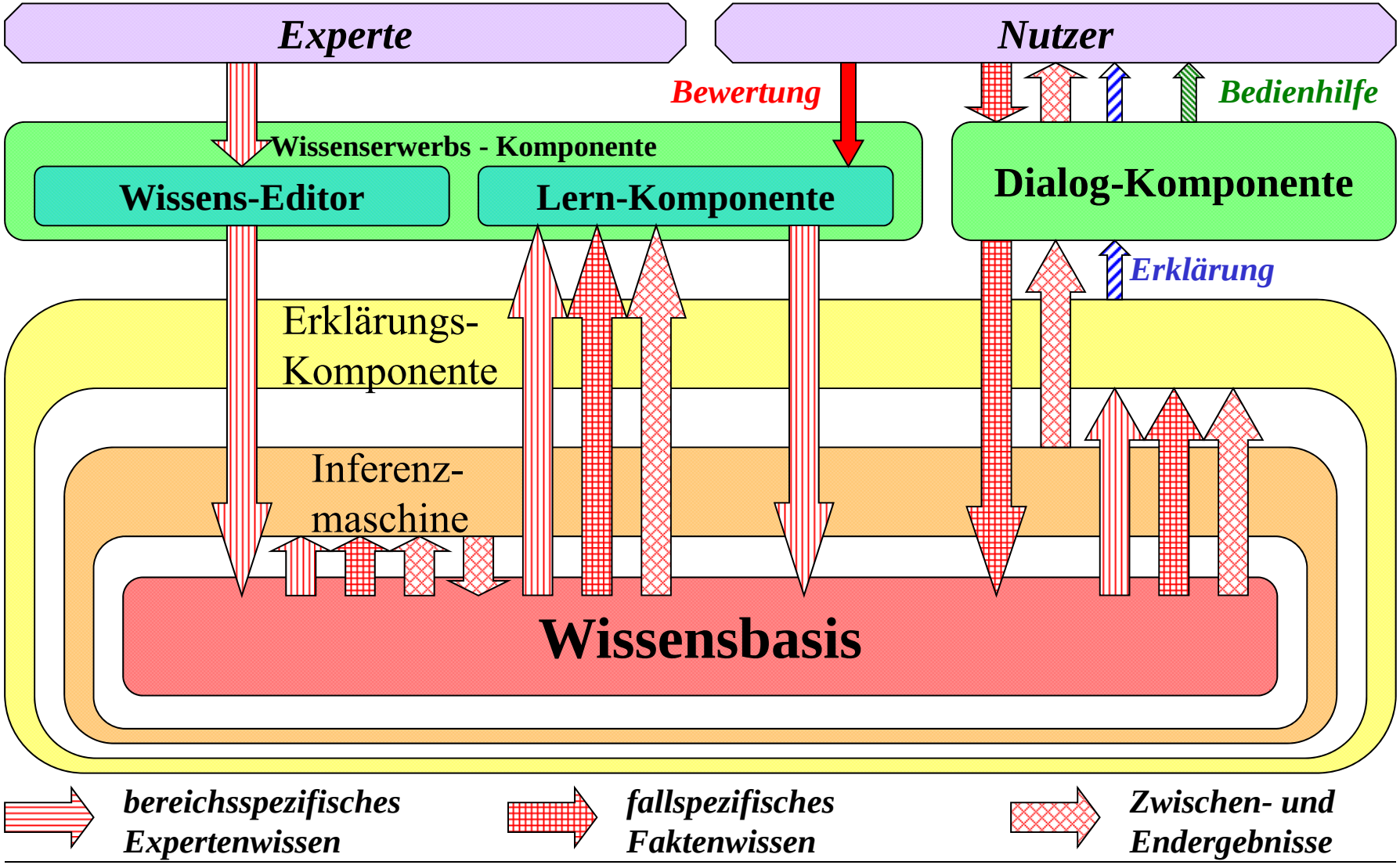
```
minrel(WB,X,Y,Z) :- member(X,WB),
                    member(Y,WB),
                    member(Z,WB),
                    min(X,Y,Z).
maxrel(WB,X,Y,Z) :- member(X,WB),
                    member(Y,WB),
                    member(Z,WB),
                    max(X,Y,Z).

min(X,Y,X) :- X <= Y.
min(X,Y,Y) :- X > Y.

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

TermauswerterMitVariablen.PRO

```
listenminrel(WB,[K|R],W) :- hat_wert(WB,K,W1),
                            minimalrel(WB,R,W1,W).
minimalrel(_,[ ],A,A).
minimalrel(WB,[K|R],A,W) :- hat_wert(WB,K,W1),
                            minrel(WB,A,W1,ANeu), minimalrel(WB,R,ANeu,W).
listenmaxrel(WB,[K|R],W) :- hat_wert(WB,K,W1),
                            maximalrel(WB,R,W1,W).
maximalrel(_,[],A,A).
maximalrel(WB,[K|R],A,W) :- hat_wert(WB,K,W1),
                            maxrel(WB,A,W1,ANeu),
                            maximalrel(WB,R,ANeu,W).
member(E,[E|_]).
member(E,[_|R]) :- member(E,R).
?- hat_wert([0,0.3,0.5,0.7,1],und(atom(X),oder(atom(0.5),atom(0.7))),Wert).
```

5 Eine Anwendung: Wissensbasierte technische Diagnose

5.1 Architektur Wissensbasierter Systeme



5 Eine Anwendung: Wissensbasierte technische Diagnose

5.2 Wissensrepräsentation

% Teilehierarchie inkl. "Relevanz" des Wissens über die Bestandteile:
teile_von(auto, [elektrik, karosse, antrieb, ...], [0.6, 0.3, 0.1, …]).
teile_von(elektrik, [anlasser, batterie, …], [0.8, 0.2, …]).)
• • •
% geschätzte Wahrscheinlichkeit für das Auftreten eines Fehlers in den Bestandteilen

% Symptombeschreibungen:
symptom(1, "keine Anlassergeräusche beim Drücken des Startknopfes", [ ]). % binär
symptom(2, "Batteriespannung in Volt", [ug(11.5)]). % numerisch: Untergrenze 11 V
symptom(3, "Leerlaufdrehzahl in U / min", [ug(900), og(1500)]). % numerisch: Unter- / Obergrenze
symptom(4, "Nach Anspringen erlöschen alle Kontrollleuchten", [ ]). % binär
• • •

% Zuornung (in Disj. Normalform) logisch verknüpfter Symptome zu Teilen inkl. einer %"Signifikanz" einer jeden Elementarkonjunktion (einer jeden Regel)
symptome_von(elektrik, [[1,-2], [-4], …], [0.8, 0.2, …]). % (1 und nicht 2) oder (nicht) 4 oder …
symptome_von(lichtmaschine, [[3,4,5], [-19,27,136], …], [0.9, 0.75, …]).
• • •
% geschätzte Wahrscheinlichkeit für die Richtigkeit der Zuordnung dieser Konjunktion zu diesem Teil

5.3 Inferenzmaschine

% Tiefensuche mit Backtrack in der Teilehierarchie
% sequenzielle Suche in der Liste der Elementarkonjunktionen der DNF

```
diagnositizierre(Teil) :-      % Hypothesengenerierung, -bestätigung, rekursive Verfeinerung
        teile_von(Teil, Teileliste. _),
        member(TeilesTeil, Teileliste),
        symptome_bestaetigt(TeilesTeil),
        diagnostiziere(TeilesTeil).


symptome_bestaetigt(T) :-
        symptome_von(T, S, _),
        member(Konjuktion, S),
        bestaetigt(Konjunktion).

bestaetigt(K) :-
        bestaetige_aus_wb(K, Rest), % bereits erhobene Symptome sind in dyn. WB
        akquiriere(Rest).           % restl. Symptome erfragen
```

5.3 Inferenzmaschine (Fortsetzung)

```
bestaetige_aus_wb([ ],[ ]).
bestaetige_aus_wb([K|R], R1) :- antw_vorhanden(K), !,
                                antwort_positiv(K),
                                bestaetige_aus_wb(R, R1).
bestaetige_aus_wb([K|R], [K|R1]) :- bestaetige_aus_wb(R, R1).
…
akquiriere([ ]).
akquiriere([K|R]) :- symptom(K,Text, Grenzen), write(Text), readln(Antwort),
                     auswerten_und_abspeichern(K, Grenzen, Antwort),
                     akquiriere(R).
auswerten_und_abspeichern(S, [ ], A): - S >= 0, !,      % pos. binäres Symptom
                     antwort_positiv(S, A), abspeichern(S, A).
auswerten_und_abspeichern(S, [ ], A): -                 % neg. binäres Symptom
                     antwort_negativ(S, A), abspeichern(S, A).
auswerten_und_abspeichern(S, [ug(U) ], W): - S>= 0, !  % pos., numerische Untergrenze
                     W > U, abspeichern(S).
…
```

5.4 Dialog- und Erklärungskomponente

Mögliche Antworten des Nutzers

1. **Beantworten der gestellten Frage:** *ja, nein, Zahlenwert, unbekannt*
2. **Aktivierung der Erklärungskomponente:**
   - *warum?*: Begründung der Fragestellung durch das System
   - *Pfad*: Auflistung der bisherigen Zwischenergebnisse (graphische Darstellung)
   - *Antworten*: Auflistung der bislang gegebenen Antworten des Nutzers
3. **Manipulation der Inferenzmaschine**
   - *zurück*: Rücksprung zu einem (auszuwählenden) Zwischenergebnis
   - *korrigieren*: Korrektur einer bereits gegebenen Antwort und Rücksprung vor diejenigen Situation, in welcher diese Antwort erstmals verwendet wurde
   - *Ende*: Abspeichern aller bislang erhobenen Daten und Zwischenergebnisse und Beendigung der Sitzung
   - *Speichern*: Abspeichern (wie oben) ohne Sitzungsende

5.5 Wissenserwerbskomponente

5.5.1 … im Dialog mit dem Autor der WB

… ist nicht mit PROLOG gemacht worden und deshalb hier kein Thema

5.5.2 … durch stochastisches Lernen

… von geschätzten Wahrscheinlichkeiten

1. … für das Auftreten eines Fehlers für jedes Element der Teilehierarchie („Relevanz“ des Diagnosewissens über dieses Teil)
2. … dafür, dass die Zuordnung einer Konjunktionen von Symptomen zu einem Teil in der Teilehierarchie sachlich richtig ist („Signifikanz“ der Diagnoseregel)

nach dem Prinzip der exponentiellen Glättung:

$$p_{i+1} := p_i + z*w$$

- p_{i+1} , p_i — geschätzte Wahrscheinlichkeit nach dem $i+1$ bzw. i – tem Lernschritt
- z — die Zufallsgröße ($z \in \{0, 1\}$)
- w — Wichtung , mit der z in die neue geschätzte Wahrscheinlichkeit eingeht ($0 < w << 1$)

Zusammenhang zwischen *w* in $p_{i+1} := p_i + z*w$ und der „Halbwertszeit“ n_h (Anzahl von Beispielen mit *z=0*, die es braucht, um die geschätzte Wahrscheinlichkeit zu halbieren:

$$(1-w)^{n_h} = 0.5 \quad \Rightarrow \quad w = 1 - 0.5^{\frac{1}{n_h}} = 1 - e^{\ln(\frac{0.5}{n_h})}$$

$$n_h = \log_{1-w} 0.5 = \frac{\ln 0.5}{\ln(1-w)}$$

n_h	w
1	*0.50000*
2	*0.29289*
5	*0.12945*
10	*0.06697*

$\Rightarrow$

Anpassen der geschätzten Wahrscheinlichkeiten (Relevanz und Signifikanz) und anschließende neue Einsortierung geänderter Werte in der Wissensbasis:

*teile_von(auto, [**elektrik**, karosse, antrieb, ...], [**0.6**, 0.3, 0.1, …]).*
*teile_von(elektrik, [anlasser, **batterie**, …], [0.8, **0.2**, …]).)*

• • •

% “geschätzte Wahrscheinlichkeiten für das Auftreten eines Fehlers in den Bestandteilen

*symptome_von(elektrik, [[1,-2], **[-4]**, …], [0.8, **0.2**, …]).*
*symptome_von(lichtmaschine, [**[3,4,5]** , [-19,27,136], …], [**0.9**, 0.75, …]).*

• • •

% geschätzte Wahrscheinlichkeiten für die Korrektheit der Zuornung einer Smptomkonjunktion

$r_{neu} = r_{alt} * (1-w) + w$	falls das zugeordnete Teil korrekt diagnostiziert wurde
$r_{neu} = r_{alt} * (1-w)$	falls das zugeordnete Teil falsch diagnostiziert wurde
$r_{neu} = r_{alt}$	ansonsten

$s_{neu} = s_{alt} * (1-w) + w$	falls das zugeordnete Teil durch diese Symptomkonjunktion korrekt diagnostiziert wurde und
$s_{neu} = s_{alt} * (1-w)$	falls das zugeordnete Teil durch diese Symptomkonjunktion alsch diagnostiziert wurde
$s_{neu} = s_{alt}$	ansonsten