Künstliche Intelligenz

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. das Skript zur Vorlesung
2. diese Foliensammlung als PowerPoint-Präsentation
3. die Übungsaufgaben zum Seminar
4. die Anleitung und Aufgabenstellung für das Praktikum
5. Hinweise zur Praktikumsdurchführung
6. die im Praktikum verwendete Entwicklungsumgebung ***Visual Prolog Personal Edition***®
7. die im Praktikum verwendete Testumgebung

Inhalt:

1. Einführung
2. Logische Grundlagen
   - Einführung in den Prädikatenkalkül der ersten Stufe (PK1)
   - Aussagen über Aussagen, Deduktion im PK1
   - Resolutionsmethode
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
4. Wissensbasierte Systeme
   - Architektur
   - Häufig verwendete Wissensrepräsentationen

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
- Wissensbasierte Systeme
- Lernen (Induktion)
- Wissensverarbeitungstechnologien (Suchtechniken, fallbasiertes Schließen, Multiagenten-Systeme)
- Sprach- und Bildverarbeitung

2 Logische Grundlagen

2.1 Einfache Aussagen

gegeben

- eine Menge von **Individuensymbole**n
- eine Menge ***n*** – stelliger **Funktionssymbole**
- eine Menge ***n*** – stelliger **Prädikatensymbole**

Term

1. Jedes Individuensymbol ist ein variablenfreier Term. (auch: Grundterm)
2. Wenn $c_1, ..., c_n$ variablenfreie Terme sind und f ein ***n*** – stelliges Funktionssymbol ist, so ist $f(c_1, ..., c_n)$ ein variablenfreier Term.
3. Weitere variablenfreie Terme gibt es nicht.

Einfache Aussage

Wenn $t_1, ..., t_n$ variablenfreie Terme sind und p ein ***n*** – stelliges Prädikatensymbol ist, so ist $p(t_1, ..., t_n)$ eine einfache Aussage.

2.2 Prädikate, Funktionen, Interpretation

Ein ***n*** - stelliges **Prädikat** bildet aus der Menge aller ***n*** - Tupel von Objekten eines Objektbereiches ***I*** eindeutig in die Menge der Wahrheitswerte ab:

$$I^n \Rightarrow \{ wahr , falsch \}$$

Eine ***n*** - stellige **Funktion** bildet aus der Menge aller ***n*** - Tupel von Objekten eines Objektbereiches ***I*** eindeutig in den Objektbereich ab:

$$I^n \Rightarrow I$$

Eine **Interpretation** ist eine Abbildung aus der Symbolwelt des PK1 in eine reale Welt:

Individuensymbole	$\Rightarrow$	***Objekte***
Prädikatensymbole	$\Rightarrow$	***Prädikate***
Funktionssymbole	$\Rightarrow$	***Funktionen***

2.3 Zusammengesetzte Aussagen

Wenn A eine Aussage ist, dann ist auch $\neg A$ eine Aussage.

Wenn A_1 und A_2 Aussagen sind, dann sind auch $(A_1 \wedge A_2)$, $(A_1 \vee A_2)$, $(A_1 \rightarrow A_2)$ und $(A_1 \leftrightarrow A_2)$ (zusammengesetzte) Aussagen.

Wahrheitswerte zusammengesetzter Aussagen

A_1	A_2	$\neg A_1$	$(A_1 \wedge A_2)$	$(A_1 \vee A_2)$	$(A_1 \rightarrow A_2)$	$(A_1 \leftrightarrow A_2)$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

2.4 Variablen und Quantifizierungen

Wenn $A(c)$ eine Aussage ist und $A(X)$ aus $A(c)$ entsteht, indem c überall durch X ersetzt wird, so sind auch $\forall X\, A(X)$ und $\exists X\, A(X)$ Aussagen.

„$\forall$“ heißt **Allquantor** , „$\exists$“ heißt **Existenzquantor** und X ist die (all- oder existenzquantifizierte) Variable.
Entsprechend der Klammerung bzw. einer Prioritätenregelung der Junktoren hat jeder Quantor seinen **Wirkungsbereich**.
Nicht quantifizierte Variablen heißen **freie Variablen**.

Für endliche Individuenbereiche $I = \{c_1, ..., c_n\}$ gilt

- $\forall$X A(X) ist äquivalent zu $\bigwedge_{i=1}^{n} A(c_i)$
- $\exists$X A(X) ist äquivalent zu $\bigvee_{i=1}^{n} A(c_i)$

2.5 Terme und Ausdrücke

Term

- Jede Variable ist ein Term.
- Jedes Individuensymbol ist ein Term.
- Wenn $t_1, \ldots, t_n$ Terme sind und f ein n – stelliges Funktionssymbol ist, dann ist $f(t_1, \ldots t_n)$ ein (strukturierter) Term.
- Weitere Terme gibt es nicht.

Ausdruck

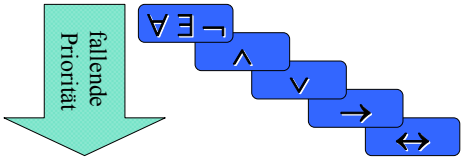
- Wenn $t_1, \ldots, t_n$ Terme sind und p ein n – stelliges Prädikatensymbol ist, dann ist $p(t_1, \ldots t_n)$ ein (atomarer) Ausdruck (eine Atomformel).
- Wenn A ein Ausdruck ist, dann ist auch $\neg A$ ein Ausdruck.
- Wenn A_1 und A_2 Ausdrücke sind, dann sind auch $(A_1 \wedge A_2)$, $(A_1 \vee A_2)$, $(A_1 \rightarrow A_2)$ und $(A_1 \leftrightarrow A_2)$ Ausdrücke.
- Wenn A ein Ausdruck und X eine Variable ist, dann sind auch $\forall X\, A(X)$ und $\exists X\, A(X)$ Ausdrücke.
- Weitere Ausdrücke gibt es nicht.

Aussagen

Ein Ausdruck ohne freie Variable heißt Aussage.

Verkürzte Notation von Ausdrücken

1. Außenklammern können weggelassen werden.
2. Ketten von Konjunktionen und Disjunktionen gelten als von links geklammert.
3. Die Stärke der Bindung der Quantoren und Junktoren ist wie folgt geregelt: $\forall$, $\exists$ und $\neg$ binden am stärksten, danach folgen (in dieser Reihenfolge) $\wedge$, $\vee$, $\rightarrow$ und $\leftrightarrow$.



2.6 Aussagen über Aussagen

Allgemeingültigkeit (Tautologien)

Eine Aussage A heißt allgemeingültig ($ag\ A$), gdw. sie für jede Interpretation wahr ist.

Kontradiktorizität

Eine Aussage A heißt kontradiktorisch ($kt\ A$), gdw. $ag\ \neg A$.

Äquivalenz von Aussagen

Zwei Aussagen A_1 und A_2 heißen äquivalent $(A_1 \equiv A_2)$, gdw. $ag\ (A_1 \leftrightarrow A_2)$.

Äquivalente Umformungen (1)

(1)	$\neg \neg A$	$\equiv$	A
(2)	$\neg (A \wedge B)$	$\equiv$	$\neg A \vee \neg B$
(3)	$\neg (A \vee B)$	$\equiv$	$\neg A \wedge \neg B$
(4)	$\neg (A \rightarrow B)$	$\equiv$	$A \wedge \neg B$
(5)	$\neg (A \leftrightarrow B)$	$\equiv$	$(A \wedge \neg B) \vee (\neg A \wedge B)$
(6)	$\neg \forall X\ A(X)$	$\equiv$	$\exists X \neg A(X)$
(7)	$\neg \exists X\ A(X)$	$\equiv$	$\forall X \neg A(X)$
(8)	$\Phi X\ A(X) \circ B$	$\equiv$	$\Phi X\ (A(X) \circ B)$

mit $\Phi \in \{ \forall , \exists \}$, $\circ \in \{ \wedge , \vee \}$, *falls X nicht in B vorkommt, es sein denn:*

(8.1)	$\forall X\ A(X) \wedge \forall X\ B(X)$	$\equiv$	$\forall X\ (A(X) \wedge B(X))$
(8.2)	$\exists X\ A(X) \vee \exists X\ B(X)$	$\equiv$	$\exists X\ (A(X) \vee B(X))$
(9)	$A \rightarrow B$	$\equiv$	$\neg A \vee B$

Äquivalente Umformungen (2)

(10)	$A \rightarrow \Phi X\ B(X)$	$\equiv$	$\Phi X\ (A \rightarrow B(X))$
(11)	$\forall X\ A(X) \rightarrow B$	$\equiv$	$\exists X\ (A(X) \rightarrow B)$
(12)	$\exists X\ A(X) \rightarrow B$	$\equiv$	$\forall X\ (A(X) \rightarrow B)$
(13)	$A \wedge (B \vee C)$	$\equiv$	$(A \wedge B) \vee (A \wedge C)$
(14)	$A \vee (B \wedge C)$	$\equiv$	$(A \vee B) \wedge (A \vee C)$
(15)	$A \wedge A$	$\equiv$	A
(16)	$A \wedge true$	$\equiv$	A
(17)	$A \wedge false$	$\equiv$	$false$
(18)	$A \vee A$	$\equiv$	A
(19)	$A \vee true$	$\equiv$	$true$
(20)	$A \vee false$	$\equiv$	A

2.7 Folgern

Sei M eine Menge von Aussagen, A eine Aussage.

A folgt aus M ($M \models A$) , falls jede Interpretation, die zugleich alle Elemente aus M wahr macht (jedes Modell von M), auch A wahr macht.

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

$M \models A$, gdw. $ag(\bigwedge_{i=1}^{n} A_i \rightarrow A)$

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

2.8 HORN-Klauseln

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

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

Varianten / Spezialfälle

1. **Regeln** (vollständige HORN-Klauseln)
   $\forall X_1 \ldots \forall X_n (A(X_1,\ldots, X_n) \leftarrow \bigwedge_{i=1}^{m} A_i(X_1,\ldots, X_n))$
2. **Fakten** (HORN-Klauseln mit leerem Klauselkörper)
   $\forall X_1 \ldots \forall X_n (A(X_1,\ldots, X_n) \leftarrow true)$
3. **Fragen** (HORN-Klauseln mit leerem Klauselkopf)
   $\forall X_1 \ldots \forall X_n (false \leftarrow \bigwedge_{i=1}^{m} A_i(X_1,\ldots, X_n))$
4. **leere HORN-Klauseln** (mit leeren Kopf & leerem Körper)
   $false \leftarrow true$

2.9 Resolution nach ROBINSON

gegeben:

- Menge von Regeln und Fakten M — $M = \{K_1,...,K_n\}$
- negierte Hypothese $\neg H$ — $\neg H \equiv \neg \bigwedge_{i=1}^{m} H_i \equiv false \leftarrow \bigwedge_{i=1}^{m} H_i$

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 Termeinsetzungen ϑ_1 und ϑ_2 in die Variablen von A und eines der H_i (etwa H_l), so dass $\vartheta_1(A) \equiv \vartheta_2(H_l)$.

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

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

$$\bigwedge_{i=1}^{l-1} \vartheta_2(H_i) \wedge \bigwedge_{k=1}^{p} \vartheta_1(B_k) \wedge \bigwedge_{i=l+1}^{m} \vartheta_2(H_i)$$

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 $false \leftarrow true \equiv \square$ ersetzt werden kann.

„Es gebe Termeinsetzungen in die Variablen von ...“

Solche Termeinsetzungen sind heißen **Unifikator** und sind Ergebnis einer

Unifikation (1)

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.

Es bleibt die Frage nach der Unifizierbarkeit von Termen:

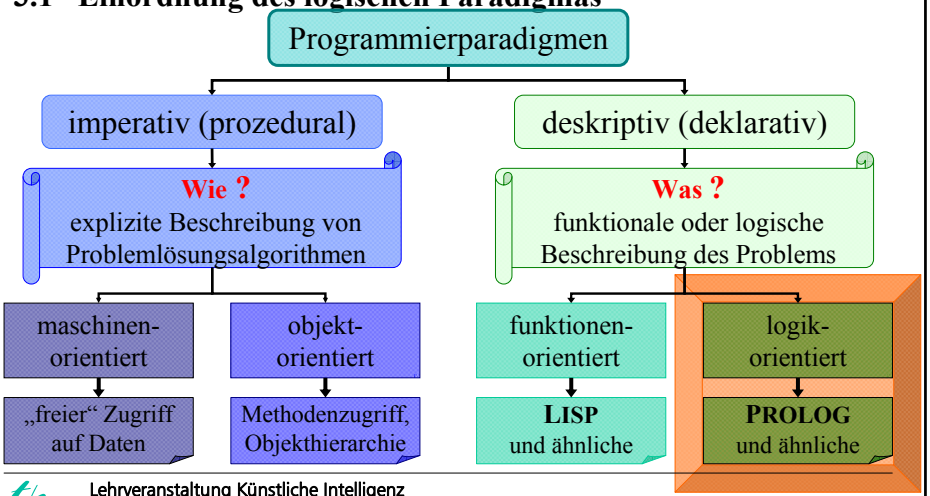
Unifikation (2)

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. Sie werden gleichgesetzt: $t_1 := t_2$ bzw. $t_2 := t_1$.

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
strukturierter 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)

• Yoshihito und Sadako sind die Eltern von Hirohito.	• vater_von(yoshihito,hirohito). mutter_von(sadako,hirohito).
• Kuniyoshi und Chikako sind die Eltern von Nagako.	• vater_von(kunioshi,nagako). mutter_von(chikako,nagako).
• Akihito‘s und Hitachi‘s Eltern sind Hirohito und Nagako.	• vater_von(hirohito,akihito). vater_von(hirohito,hitachi). mutter_von(nagako,akihito). mutter_von(nagako,hitachi).
• Der Großvater ist der Vater des Vaters oder der Vater der Mutter.	• 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 haben den gleichen Vater und die gleiche Mutter.	• geschwister(X,Y) :- vater_von(V,X), vater_von(V,Y), mutter_von(M,X), mutter_von(M,Y).

3.3.1 Formulierung von Wissensbasen (2)

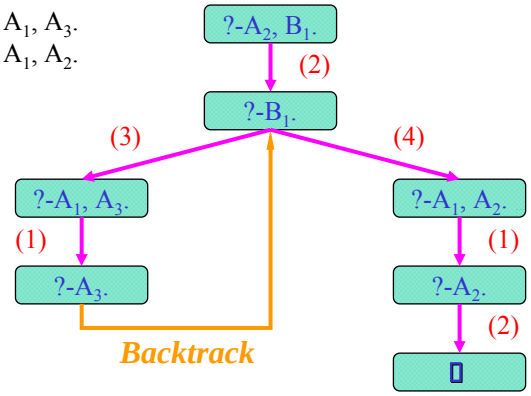
• Kollegen Meier und Müller arbeiten im Raum 1, Kollege Otto im Raum 2 und Kollege Kraus im Raum 3.	• arbeitet_in(meier,raum_1). arbeitet_in(mueller,raum_1). arbeitet_in(otto,raum_2). arbeitet_in(kraus,raum_3).
• Netzanschlüsse gibt es in den Räumen 2 und 3.	• anschluss_in(raum_2). anschluss_in(raum_3).
• Ein Kollege ist erreichbar, wenn er in einem Raum mit Netzanschluss arbeitet.	• erreichbar(K) :- arbeitet_in(K,R), anschluss_in(R).
• 2 Kollegen können Daten austauschen, wenn sie im gleichen Raum arbeiten oder beide erreichbar sind.	• koennen_daten_austauschen(K1,K2) :- arbeitet_in(K1,R), arbeitet_in(K2,R). koennen_daten_austauschen(K1,K2) :- erreichbar(K1), erreichbar(K2).

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 ...

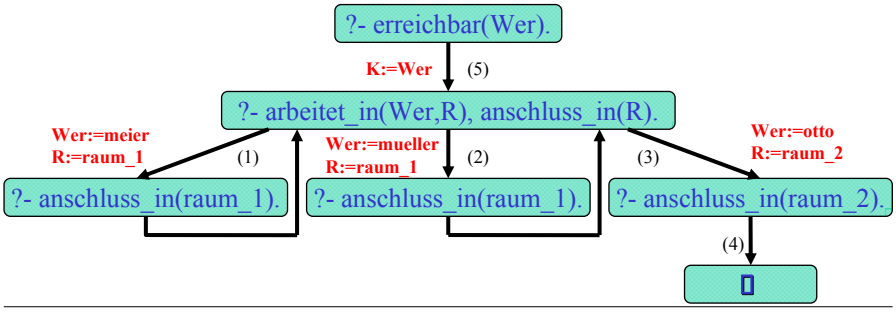
- ... genau eine, so wird das 1. Teilziel durch deren Körper ersetzt.
- ... mehrere, so wird das aktuelle Ziel inklusive alternativ anwendbarer Klauseln im Backtrack-Keller abgelegt und die am weitesten oben stehende Klausel angewandt.
- ... keine (mehr), so wird mit dem auf dem Backtrack-Keller liegendem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

Prädikate zur Steuerung der Suche nach einer Folge von Resolutionsschritten

! (cut)

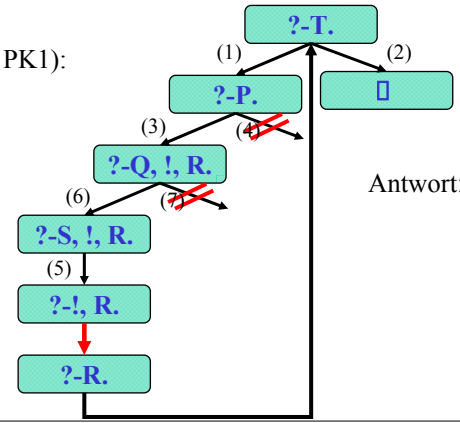
Das Prädikat !/0 ist stets wahr.
In Klauselkörpern eingefügt verhindert es ein Backtrack der hinter !/0 stehenden Teilziele zu den vor !/0 stehenden Teilzielen sowie zu alternativen Klauseln des gleichen Kopfprädikats.
Die Verarbeitung von !/0 schneidet demnach alle vor der Verarbeitung verbliebenen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

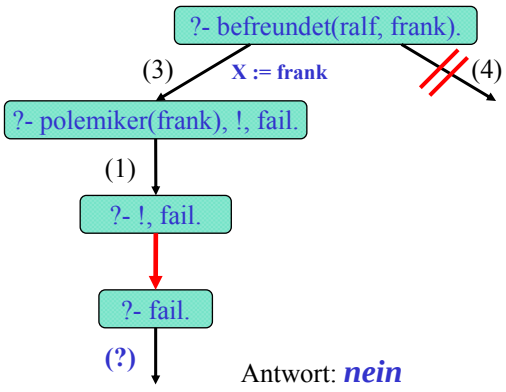
Das Prädikat fail/0 ist stets falsch.
In Klauselkörpern eingefügt löst es ein Backtrack aus bzw. führt zum Misserfolg, falls der Backtrack-Keller leer ist, d.h. falls es keine verbleibenden Lösungswege (mehr) gibt.

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

wieder ‘n Beispiel:

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

?- befreundet(ralf, frank).

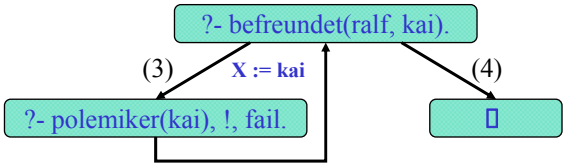


Antwort: ***nein***

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

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.

   ungebräuchlich

   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.

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).
```


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 eine 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* die Liste *D* ist:**
   [E|R] – L = D *, wenn* E ∈ L *und* R – L = D
3. **Die Differenz aus einer Liste *[E|R]* und einer Liste *L* , welche *E* nicht ent-hält, 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)

```
(1) interpret( [ [ ] , _ ] , [ ] ) .
(2) interpret( [ [E|R] , L] , D ) :- member(E , L), ! , interpret( [ R , L ] , D ) .
(3) interpret( [ [E|R] , L] , [E|D] ) :- interpret( [ R , L ] , D ) .
```

Differenzlisten: *Eine Anwendung*

```
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

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:

```
(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*

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:

```
(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:

```
(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*

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:

```
(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

Wider-spruch

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

4 Typische Problemklassen für die Anwendung der Logischen Programmierung

4.1 Rekursive Problemlösungsstrategien

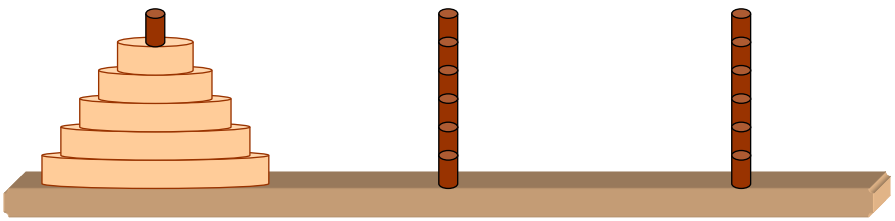
Botschaft # 1

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

```
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** aus dem Vokabular abzüglich des Alphabets: $N = V \setminus A$
3. einer Menge von Ableitungsregeln $R \subseteq N \times (N \cup A)^*$
4. dem Satzsymbol $S \in N$

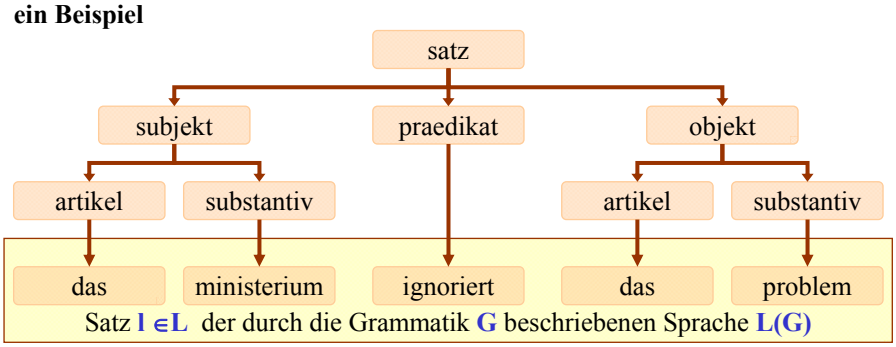
... und in PROLOG repräsentiert werden durch

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 $l \in N$ im Kopf und $r \in (N \cup A)^*$ 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

```
% 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

```
% 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

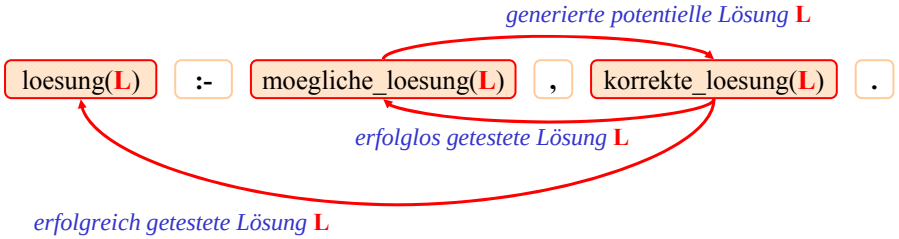
1. ***eine potentielle Lösung zu „erraten“ und dazu***
2. ***ein Verfahren zu entwickeln, welches diese Lösung auf Korrektheit testet,***

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:

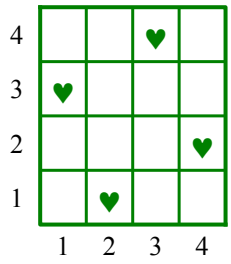
- Besteht **L** diesen Korrektheitstest, ist eine Lösung gefunden.
- Fällt **L** bei diesem Korrektheitstest durch, wird mit Backtrack das Prädikat moegliche_loesung(L) um eine alternative potentielle Lösung ersucht.



ein Beispiel

konfliktfreie Anordnung von *N* Damen auf einem *N×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) **Variante:**
  Liste der Spaltenindizes
  [Spalte_zu_Zeile_1, ..., Spalte_zu_Zeile_N]
  [2, 4, 1, 3]


```
(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 ] ] , I ) :-
         E2 = E1 + I, ! , fail .
(12) teste( [ E1 | [ E2 | R ] ] , I ) :-
         E2 = E1 – I , ! , fail .
(13) teste( [ E1 | [ E2 | R ] ] , I ) :-
         I1 = I + 1 ,
         teste( [E1 | R ] , I1 ),
         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)**

```
% 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 ) :-
   ! , 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

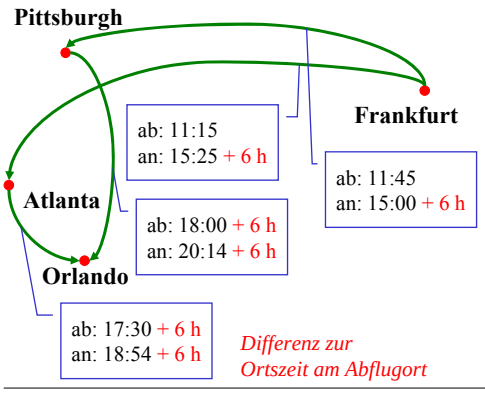
Botschaft # 5

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.

Die Suchstrategie

```
start(S,Z,Zeit,Liste) :- suche(S,Z,Zeit,Liste), vergleiche(Liste), fail .
start(S,Z,_,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(S,Z1,Z,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 .
```