Programmierparadigmen der Künstlichen 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 ein 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
4. Funktionale Programmierung
   - Einführung
   - LISP-Ausdrücke
   - Elementare LISP-Funktionen
   - Komplexere LISP-Funktionen
   - Ein Beispiel: Tiefensuche in Graphen

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 **Individuensymbolen**
- 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 $\boldsymbol{f(c_1, ..., c_n)}$ 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 $\boldsymbol{p(t_1, ..., t_n)}$ 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:

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

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

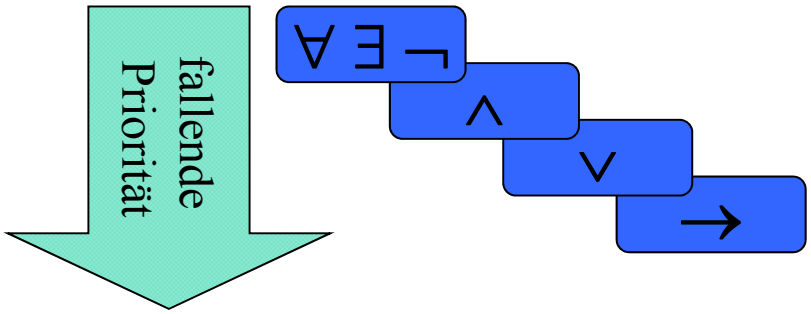
- Wenn $t_1, \ldots, t_n$ Terme sind und p ein n – stelliges Prädikatensymbol ist, dann ist $p(t_1, \ldots t_n)$ ein (atomarer) Ausdruck (eine Atomformel).
- Wenn A ein Ausdruck ist, dann ist auch $\neg A$ ein Ausdruck.
- Wenn A_1 und A_2 Ausdrücke sind, dann sind auch $(A_1 \wedge A_2)$, $(A_1 \vee A_2)$, $(A_1 \rightarrow A_2)$ 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)	$A \wedge \mathit{true}$	$\equiv$	A
(17)	$A \wedge \mathit{false}$	$\equiv$	false
(18)	$A \vee A$	$\equiv$	A
(19)	$A \vee \mathit{true}$	$\equiv$	true
(20)	$A \vee \mathit{false}$	$\equiv$	A

2.7 Folgern

Sei ***M*** eine Menge von Aussagen, ***A*** eine Aussage.

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

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

M |= ***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

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

$A(X_1,..., X_n)$, $A_i(X_1,..., X_n)$
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)

$$false \leftarrow true$$

2.9 Resolution nach ROBINSON

gegeben:

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

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

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

Ziel:

- Beweis, dass $M \models H$

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

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 $false \leftarrow true \equiv$ 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:

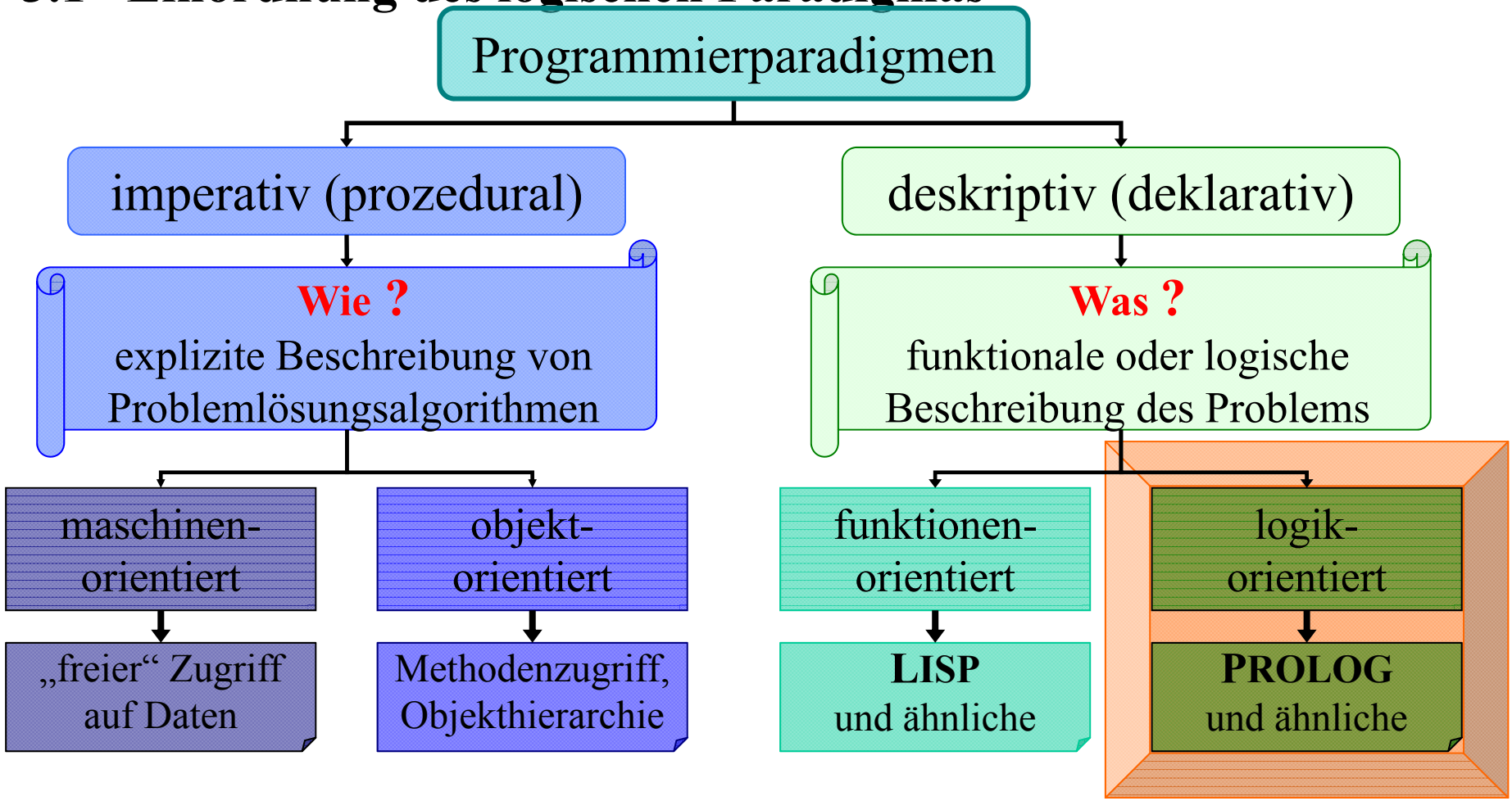
Unifikation (2)

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 $f_1(t_{11} , ... , t_{1n})$ und $f_2(t_{21} , ... , t_{2m})$ 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
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)

- 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).

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.
- 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).

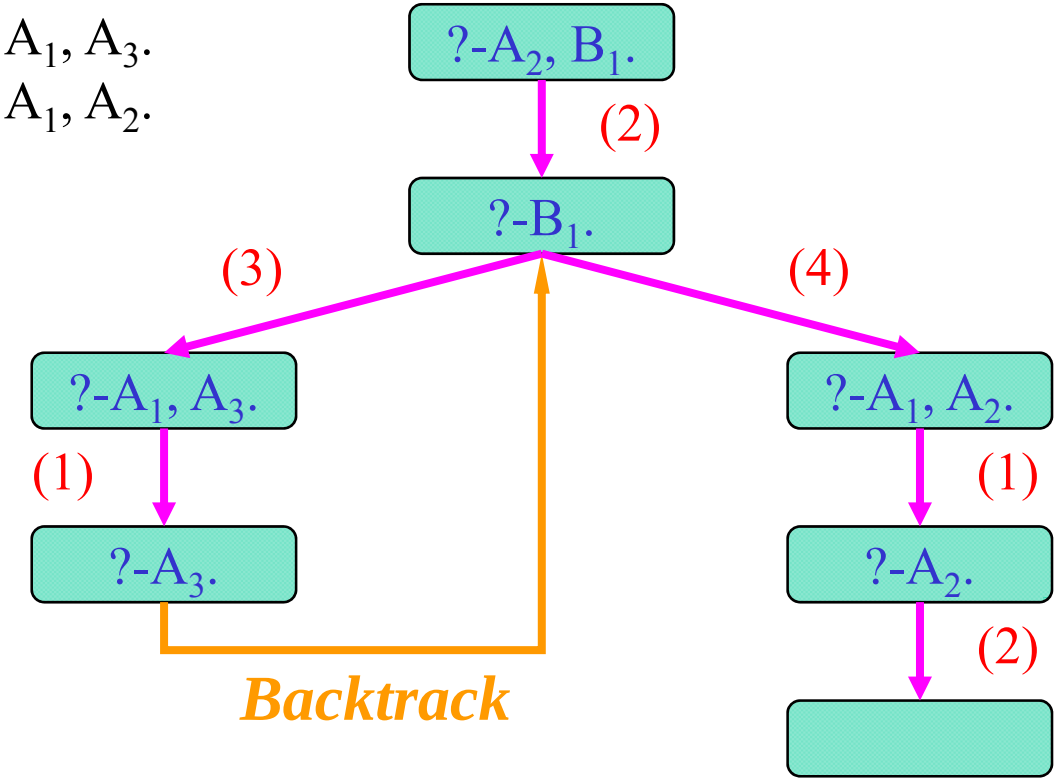
Wissensbasis:

(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 ...

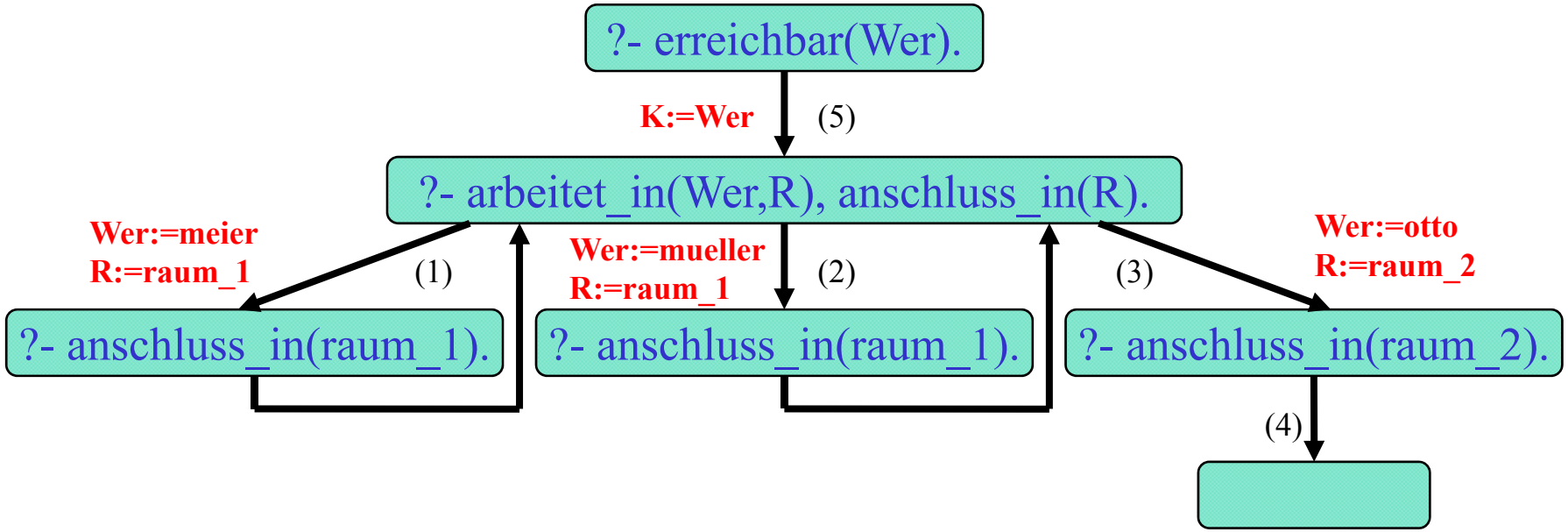
- ... genau eine, so wird das 1. Teilziel durch deren Körper ersetzt.
- ... mehrere, so wird das aktuelle Ziel inklusive alternativ anwendbarer Klauseln im Backtrack-Keller abgelegt und die am weitesten oben stehende Klausel angewandt.
- ... keine (mehr), so wird mit dem auf dem Backtrack-Keller liegendem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

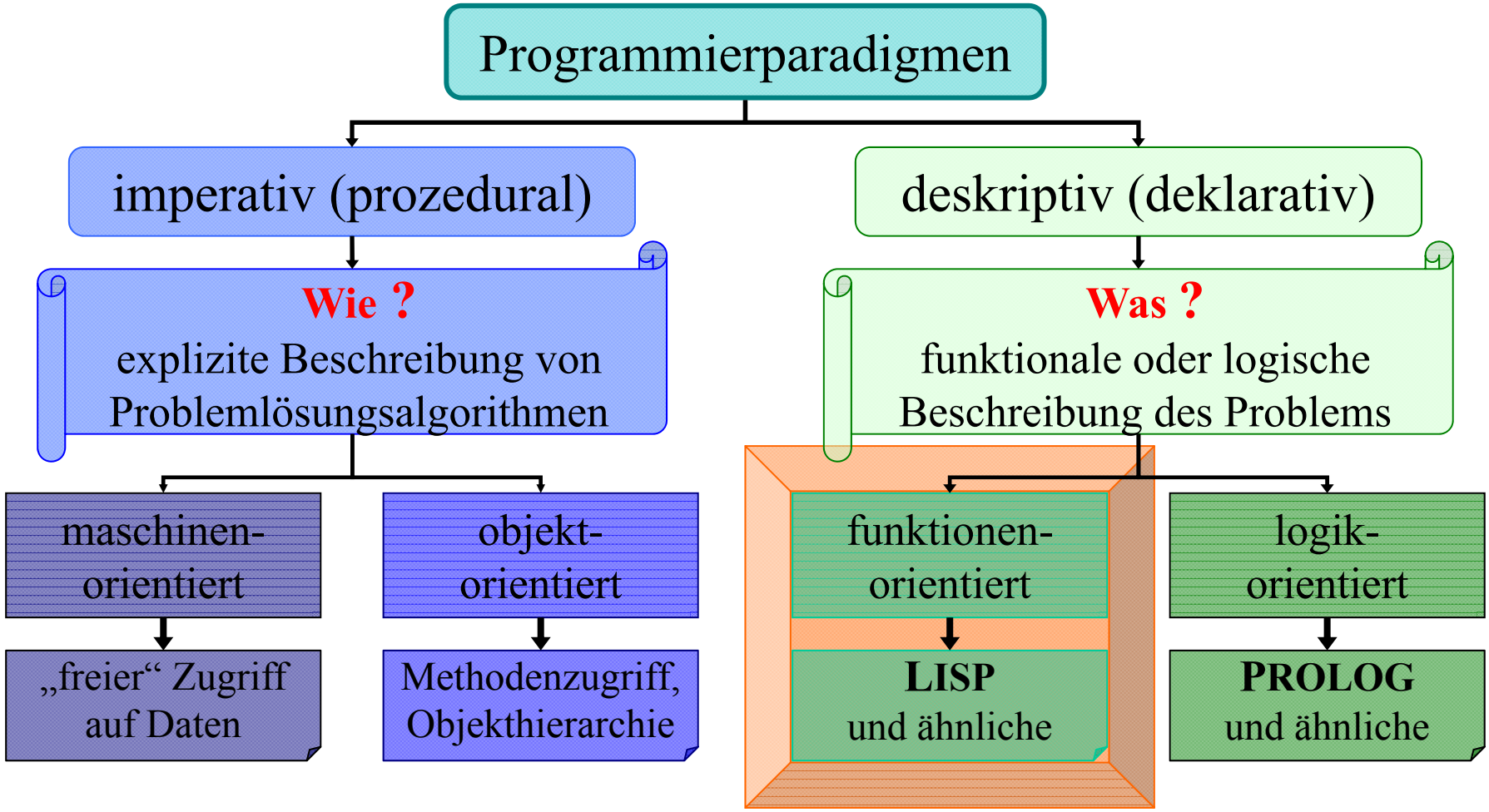
(1) arbeitet_in(meier,raum_1).
(2) arbeitet_in(mueller,raum_1).
(3) arbeitet_in(otto,raum_2).
(4) anschluss_in(raum_2).
(5) erreichbar(K) :- arbeitet_in(K,R), anschluss_in(R).

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



4 Funktionale Programmierung

4.1 Einordnung des funktionalen Paradigmas



4.1 Einführung in die Funktionale Programmierung

LISt **P**rocessor: ***Eine Sprache zur Verarbeitung von Symbolen in Listen***

Sprachen vor der Entstehung von LISP (Ende 50er Jahre) dienten ausschließlich

- **numerischen oder systemverwaltenden Aufgaben**;

die theoretische Informatik und die entstehende KI verlangte jedoch eine

- **symbolverarbeitende Sprache**.

Entstehungsgeschichte

Theoretische Informatiker (Alan TURING [1912-1954, siehe www.turing.org.uk/turing] u.a.) begannen, den Berechenbarkeitsbegriff zu erfassen und schufen

- Modelle (Stichwort: *TURING-Maschine*) und
- Definitionen (Stichworte: *Grundfunktionen, Verschachtelung, Projektion, partielle Rekursion*),

welche die Berechenbarkeit von Funktionen $f: IN^n \rightarrow IN$ modellieren.

Alles Berechenbare auch programmierbar zu machen, erfordert demnach eine Sprache, welche exakt dieser Definition folgt, d.h. die Notation der Grundfunktionen, Verschachtelung, Projektion und partielle Rekursion erlaubt: **LISP**.

- Funktional programmieren heißt, über einem Symbolvorrat (Vokabular) $\boldsymbol{V}$ Funktionen

$$\boldsymbol{f: V^n \rightarrow V}$$

zu notieren.

- Der Funktionsname wird durch einen OPERATOR spezifiziert. Sie wird zusammen mit ihren Argumenten, den OPERANDen in Form einer Liste in „polnischer“ Notation aufgerufen:

$$\mathbf{(OPERATOR \quad OPERAND_1 \quad OPERAND_2 \quad \ldots \quad OPERAND_n)}$$

- Ihr Funktionswert wird durch Auswertung in einer „Read-Eval-Print Loop“ ermittelt, welche
  1. derart notierte Ausdrücke einliest,
  2. diese (rekursiv) auswertet (d.h. Operanden können selbst Funktionsaufrufe sein) und
  3. das Ergebnis der Auswertung ausdruckt.

  Ein Beispiel aus der Arithmetik:
  - (+ 7 (* 3 4))

  19

4.2 LISP - Ausdrücke

4.2.1 Syntax

4.2.1.1 Literale Atome

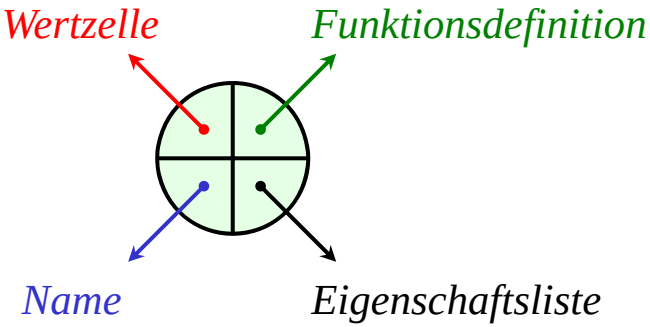
Literale Atome sind Folgen von Buchstaben und Ziffern, deren erstes Zeichen ein Buchstabe ist.

Die Atome **`NIL`** (*false* , *leere Liste*) und **`T`** (*true*) sind spezielle literale Atome.

Verwendung

- als Datum
- als Variablenbezeichnung
- als Funktionsbezeichnung

Repräsentation



4.2.1.2 Numerische Atome

Numerische Atome sind ganze Zahlen oder Gleitkommazahlen.

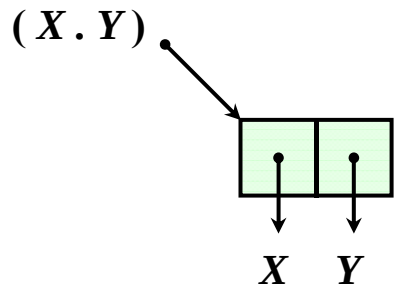
4.2.1.3 Symbolische Ausdrücke

1. Literale Atome sind symbolische Ausdrücke.
2. Numerische Atome (Zahlen) sind symbolische Ausdrücke.
3. Wenn ***X*** und ***Y*** symbolische Ausdrücke sind, so ist das gepunktete Paar **(*X* . *Y*)** ein symbolischer Ausdruck.
4. Weiter symbolische Ausdrücke gibt es nicht.

Verwendung

- als Datenstruktur

Repräsentation



4.2.1.4 Listen

1. **`NIL`** ist die leere Liste; sie kann alternativ als **`[ ]`** notiert werden.
2. Verschachtelte gepunktete Paare der Form

   $$(E_1 . (E_2 . (E_3 \ldots (E_n . \texttt{NIL}) \ldots)))$$

   bilden eine nichtleere Liste und können alternativ als

   $$(E_1\ E_2\ E_3\ \ldots\ E_n)$$

   notiert werden.
3. Weitere Listen gibt es nicht.

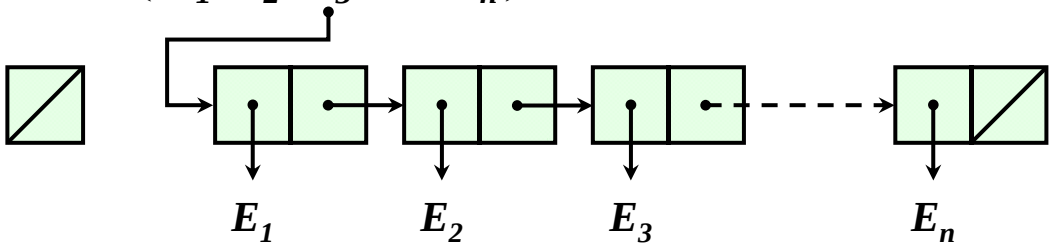
Nichtleere Listen sind Spezialfälle gepunkteter Paare.
Die leere Liste ist spezielles literales Atom.

Verwendung

- als Datenstruktur
- zum Funktionsaufruf
- zur Funktionsdefinition

Repräsentation

`NIL` $(E_1\ E_2\ E_3\ \ldots\ E_n)$



4.2.2 Auswertung von Ausdrücken

Auswerteregeln

(1) Der Wert eines numerischen Atoms (einer Zahl) ist diese Zahl selbst.

(2) Der Wert eines literalen Atoms ist das Objekt, welches bei der Auswertung des durch die Wertzelle referenzierten Ausdrucks entsteht.

(3) Der Wert eines Ausdrucks **(OPERATOR $OPERAND_1$ … $OPERAND_n$)** ist derjenige Funktionswert, welcher entsteht, wenn man
- die Operanden **$OPERAND_1$ … $OPERAND_n$** von links nach rechts (nach denselben Regeln) auswertet und
- auf die Ergebnisse den Operator **OPERATOR** anwendet.

(4) Der Wert eines Ausdrucks **(SETQ $OPERAND_1$ $OPERAND_2$)** ist der Wert des zweiten Operanden. Die Auswertung von **SETQ** hat den Nebeneffekt, dass der Wert von **$OPERAND_2$** in die Wertzelle von **$OPERAND_1$** eingetragen wird.

(5) Der Wert eines Ausdrucks **(QUOTE OPERAND)** bzw. **(' OPERAND)** ist gleich dem unausgewerteten Operanden, d.h. gleich seinem Namen bzw. des aus Namen gebildeten Ausdrucks.

4.3 Elementare LISP - Funktionen

4.3.1 Zugriffsfunktionen

1. Der Wert eines Ausdrucks **(CAR LISTE)** ist der Wert des ersten Elements der Liste **LISTE** .
2. Der Wert eines Ausdrucks **(CDR LISTE)** ist die Restliste, d.h. die Liste der Elemente nach dem ersten Element der Liste **LISTE** .

Übliche Schreibweisen für Verschachtelungen: **C** <Folge von **A**s und **D**s> **R** , z.B.

- **(CAAR LISTE)** steht für **(CAR (CAR LISTE))**
- **(CDAR LISTE)** steht für **(CDR (CAR LISTE))**
- **(CADR LISTE)** steht für **(CAR (CDR LISTE))**
- **(CADAR LISTE)** steht für **(CAR (CDR (CAR LISTE)))**
- **(CAADR LISTE)** steht für **(CAR (CAR (CDR LISTE)))**
- **(CADAAR LISTE)** steht für **(CAR (CDR (CAR (CAR LISTE))))**

4.3.2 Konstruktionsfunktionen

1. Der Wert eines Ausdrucks **(CONS $OPERAND_1$ $OPERAND_2$)** ist das gepunktete Paar **($OPERAND_1$. $OPERAND_2$)** .
2. Der Wert eines Ausdrucks **(LIST $OPERAND_1$ $OPERAND_2$ … $OPERAND_N$)** ist die Liste **($OPERAND_1$ $OPERAND_2$ … $OPERAND_N$)** .

4.3.3 Typ-Prädikate

Vorbemerkung: **NIL** hat zwei Interpretationen: ***leere Liste*** und ***false***

4.3.3.1 Typ-Erkennung

1. Der Wert eines Ausdrucks **(ATOM OPERAND)** ist
   - **T** (*true*) , falls der Wert von **OPERAND** ein (literales oder numerisches) Atom ist bzw.
   - **NIL** (*false*) anderenfalls.
2. Der Wert eines Ausdrucks **(NUMBERP OPERAND)** ist
   - **T** (*true*) , falls der Wert von **OPERAND** ein numerisches Atom ist bzw.
   - **NIL** (*false*) anderenfalls.

3. Der Wert eines Ausdrucks **(LISTP OPERAND)** ist
   - **T** (*true*) , falls der Wert von **OPERAND** eine nichtleere Liste ist bzw.
   - **NIL** (*false*) anderenfalls.
4. Der Wert eines Ausdrucks **(NULL OPERAND)** ist
   - **T** (*true*) , falls der Wert von **OPERAND NIL** (*false, leere Liste*) ist bzw.
   - **NIL** (*false*) anderenfalls.

4.3.3.2 Gleichheits-Erkennung

Man unterscheidet (1.) Gleichwertigkeit (Äquivalenz) und (2.) syntaktische Gleichheit (Uniformität).

1. Der Wert eines Ausdrucks **(EQUAL LISTE_1 LISTE_2)** ist
   - **T** (*true*) , falls die Listen **LISTE_1** und **LISTE_2** strukturell und wertmäßig gleich (d.h. äquivalent) sind bzw.
   - **NIL** (*false*) anderenfalls.
2. Der Wert eines Ausdrucks **(EQL LISTE_1 LISTE_2)** ist
   - **T** (*true*) , falls die Listen **LISTE_1** und **LISTE_2** syntaktisch gleich sind bzw.
   - **NIL** (*false*) anderenfalls.

4.4 Definition von LISP - Funktionen

1. **(LAMBDA PARAMETERLISTE AUSDRUCK)** wendet die im LISP-Ausdruck **AUSDRUCK** beschriebene Funktion auf die Liste **PARAMETERLISTE** an.
   Das Ergebnis dieser Anwendung ist der Funktionswert.

2. **(DEFUN NAME PARAMETERLISTE AUSDRUCK)** gibt einer
   - im LISP-Ausdruck **AUSDRUCK** beschriebenen Funktion
   - mit den in der Liste **PARAMETERLISTE** notierten symbolischen Parametern
   - den in **NAME** spezifizierten Namen,

   unter welchem diese Funktion aufrufbar ist. Der Funktionswert ist **T** (*true*).

3. **(FUNCALL FUNKTION ARGUMENTE)** wendet
   - die durch die Auswertung des literalen Atoms **FUNKTION**, dessen Wertzelle auf einen LAMBDA-Ausdruck (siehe 1.) verweist, ermittelte Funktion
   - auf die Liste **ARGUMENTE** an.

   Das Ergebnis dieser Anwendung ist der Funktionswert.

4. **(APPLY FKTNAME ARGUMENTE)** ermittelt
   - den Namen einer Funktion durch Auswertung des literalen Atoms **FKTNAME**
   - und wendet diese Funktion auf die Liste **ARGUMENTE** an.

   Das Ergebnis dieser Anwendung ist der Funktionswert.

Erläuterung

1. **(LAMBDA PARAMETERLISTE AUSDRUCK)**
   definiert eine Funktion, wendet sie an und „vergisst“ sie augenblicklich wieder.
2. **(DEFUN NAME PARAMETERLISTE AUSDRUCK)**
   definiert eine Funktion und verleiht ihr einen Namen, unter welchem sie zukünftig aufrufbar ist.
3. **(FUNCALL FUNKTION ARGUMENTE)**
   „errechnet“ die Definition einer Funktion, die nicht vordefiniert ist, und wendet diese Funktion an. **FUNCALL** verwendet man typischerweise, wenn gleichbenannte Funktionen verschiedene Definitionen haben sollen.
   Hiermit kann man z.B. ähnliche Operationen für verschiedene Datentypen (etwa die Multiplikation von Skalaren, Matrizen, Mengen, …) identisch (etwa ∗) nennen.
4. **(APPLY FKTNAME ARGUMENTE)**
   „errechnet“ den Namen einer Funktion, die vordefiniert ist, und wendet diese Funktion an. **APPLY** verwendet man typischerweise, wenn die anzuwendende Funktion im Kontext von Bedingungen ermittelt werden muss.
   Hiermit kann man z.B. für Operationen (etwa ∩ oder ∧) , für die in verschiedenen Kontexten (etwa Mengenalgebra und Logik) gleiche Gesetzmäßigkeiten gelten, diese Gesetze kontextübergreifend notieren.

4.5 Komplexere LISP - Funktionen

4.5.1 Listenverarbeitung

1. Der Wert eines Ausdrucks **(APPEND LISTE$_1$ LISTE$_2$)** ist diejenige Liste, die sich durch das Anhängen der Liste **LISTE$_2$** and die Liste **LISTE$_1$** ergibt.
2. Der Wert eines Ausdrucks **(REVERSE LISTE)** ist diejenige Liste, die sich durch das Umkehren der Reihenfolge der Elemente aus der Liste **LISTE** ergibt.
3. Der Wert eines Ausdrucks **(DELETE ELEMENT LISTE)** ist diejenige Liste, die sich ergibt, wenn man alle Elemente, die den gleichen Wert wie **ELEMENT** haben, aus der Liste **LISTE** entfernt.
   Die Löschung wird auch physisch ausgeführt, d.h. das Argument LISTE verändert bei der Ausführung dieser Funktion seinen Wert.
4. Der Wert eines Ausdrucks **(MEMBER ELEMENT LISTE)** ist
   - diejenige Liste, die sich ergibt, wenn man das erste Element in der Liste **LISTE**, welches den gleichen Wert wie **ELEMENT** hat, aus der Liste **LISTE** entfernt bzw.
   - **NIL**, falls kein derartiges Element in dieser Liste vorkommt.

5. Der Wert eines Ausdrucks **(RPLACA LISTE NEUES_ELEMENT)** ist diejenige Liste, die sich aus der Liste **LISTE** ergibt, wenn man deren erstes Element durch das Element **NEUES_ELEMENT** austauscht.

   Die Löschung wird auch physisch ausgeführt, d.h. das Argument LISTE verändert bei der Ausführung dieser Funktion seinen Wert.

6. Der Wert eines Ausdrucks **(RPLACD LISTE NEUE_RESTLISTE)** ist diejenige Liste, die sich aus der Liste **LISTE** ergibt, wenn man deren Restliste durch die Liste **NEUE_RESTLISTE** austauscht.

   Die Löschung wird auch physisch ausgeführt, d.h. das Argument LISTE verändert bei der Ausführung dieser Funktion seinen Wert.

4.5.2 Bedingte Anweisungen

Auswerteregeln *(Nachtrag einer neuen Regel zu denen aus Abschnitt 2.2, Folie #10)*

(6) Zur Ermittlung des Wertes eines Ausdrucks

(COND (RPÄDIKAT$_1$ AUSDRUCK$_{11}$ … AUSDRUCK$_{1N_1}$)
(RPÄDIKAT$_2$ AUSDRUCK$_{21}$ … AUSDRUCK$_{2N_2}$)
…
(RPÄDIKAT$_M$ AUSDRUCK$_{M1}$ … AUSDRUCK$_{MN_M}$))

werden die Prädikate **PRÄDIKAT$_1$ … PRÄDIKAT$_M$** in dieser Reihenfolge so lange ausgewertet, bis sich für eines der Prädikate ein von **NIL** verschiedener Wert ergibt oder alle Prädikate ausgewertet wurden.

(6.1) Wird ein solches Prädikat gefunden, werden die in der gleichen Liste stehenden Ausdrücke in der notierten Reihenfolge ausgewertet. Der Wert des letzten dieser Ausdrücke ist der Funktionswert des **COND**-Ausdrucks.
Enthält diese Liste keine Ausdrücke, ist der Wert des Prädikats der Funktionswert des **COND**-Ausdrucks.

(6.2) Hat keines der Prädikate einen von **NIL** verschiedenen Wert, ist **NIL** der Funktionswert des **COND**-Ausdrucks.

5.3 LET - Konstruktionen

(LET ((VARIABLE$_1$ AUSDRUCK$_1$)
(VARIABLE$_2$ AUSDRUCK$_2$)
…
(VARIABLE$_N$ AUSDRUCK$_N$))
LET_KÖRPER)

Diese Konstruktion der Definition lokaler Variablen:

- **VARIABLE$_1$ … VARIABLE$_N$** sind literale Atome, welche durch Auswertung des jeweils nachstehenden Ausdrucks einen Wert erhalten.
- Im **LET_KÖRPER** können dann die Variablen anstelle ihres korrespondierenden Ausdrucks verwendet werden.

Die Verwendung dieser Konstruktion verbessert

a) die Beschreibungskomplexität, da diese Ausdrücke bei mehrfacher Verwendung nur einmal notiert werden,

b) die Speicherkomplexität, da diese Ausdrücke bei mehrfacher Verwendung nur einmal repräsentiert werden, und

c) die Laufzeitkomplexität, da diese Ausdrücke bei mehrfacher Verwendung nur einmal ausgewertet werden.

4.6 Ein Beispiel: Tiefensuche in Graphen

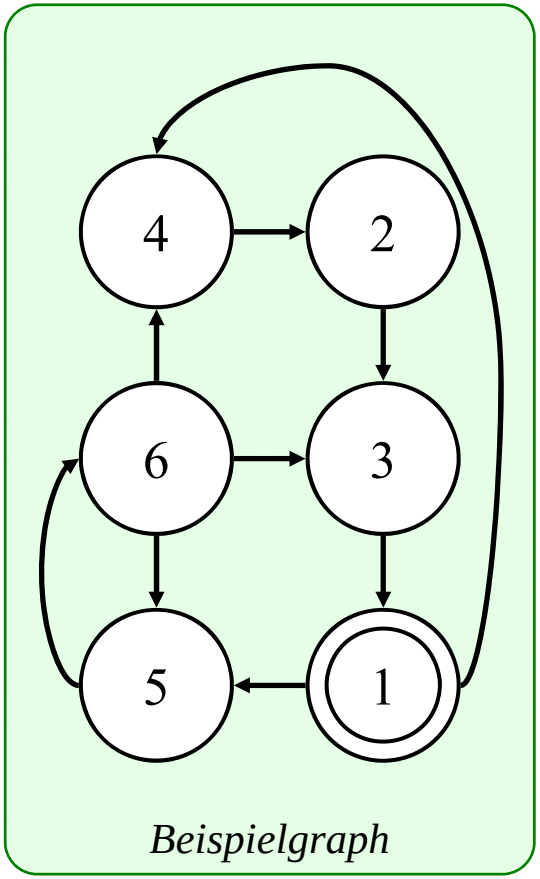
gegeben
- gerichteter Graph ***G***
- Startknoten ***S*** in ***G***

gesucht
- ein Erreichbarkeitsbaum ***B*** des Graphen ***G*** mit der Wurzel ***S***

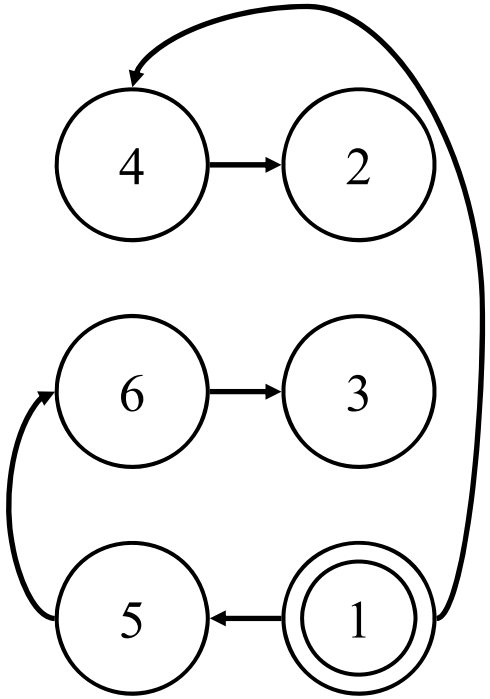
Was ist ein Erreichbarkeitsbaum?

Ein Erreichbarkeitsbaum ***B*** eines gerichteten Graphen ***G*** mit einem Startknoten ***S*** ist ein Teilgraph von ***G***, welcher folgenden Bedingungen genügt:

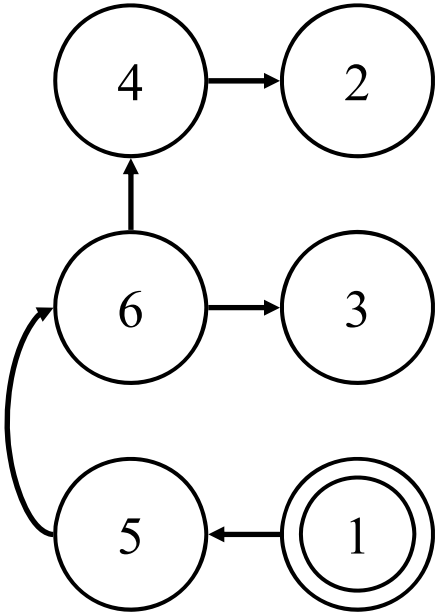
1. Die Wurzel von B ist S.
2. Alle Knoten aus G, welche von S erreichbar sind, sind in B enthalten.
3. B ist wirklich ein Baum, d.h. jeder Knoten K, der verschieden von S ist, ist auf genau einem Wege von S erreichbar.



Beispielgraph



ein Erreichbarkeitsbaum



noch ein Erreichbarkeitsbaum

Repräsentation als Liste von Kanten

```
(SETQ   BSPGRAPH
        ((2 3)(1 5)(3 1)(1 4)(5 6)(4 2)(6 5)(6 3)(6 4)))
```

Bestimmung der ersten bzw. nächsten Kante zu einem Restgraphen, welche von einem gegebenen Knoten in einem gegebenen Graphen ausgeht

```
(DEFUN NAECHSTE_KANTE (GRAPH STARTKNOTEN)
   (COND     ((NULL GRAPH)                        NIL)
             ((EQUAL STARTKNOTEN (CAAR GRAPH)) GRAPH)
             ( T
                  (NAECHSTE_KANTE (CDR GRAPH STARTKNOTEN))))
```

Diese Funktion

- liefert **NIL**, falls der Graph leer ist
- liefert den gesamten Graphen **GRAPH**, falls gleich die erste Kante von diesem Startknoten ausgeht
- sucht anderenfalls ab der zweiten Kante rekursiv weiter

Test auf Vorkommen eines Knotens in einem gegebenen Baum

```
(DEFUN ENTHALTEN (KNOTEN BAUM)
   (COND     ((NULL BAUM)                        NIL)
             ((EQUAL KNOTEN (CADAR BAUM))        T)
             ( T
                                  (ENTHALTEN KNOTEN (CDR BAUM)))))
```

Suche einer in den Erreichbarkeitsbaum aufzunehmenden neuen Kante

```
(DEFUN NEUE_KANTE (GR STKN BAUM)
 (COND ((NULL GR)    NIL)
       ( T (LET ((RESTGR (NAECHSTE_KANTE GR STKN)))
           (COND ((ENTHALTEN (CADAR RESTGR) BAUM)
                             (NEUE_KANTE (CDR RESTGR) STKN BAUM))
                 ( T      RESTGR))))))
```

Diese Funktion sucht im Graphen **GR** die erste Kante, die am Startknoten **STKN** beginnt und deren Zielknoten noch nicht in **BAUM** enthalten ist.
Findet sie eine solche, liefert sie **RESTGR**, anderenfalls **NIL**.

Die nachfolgende Funktion **SUCHSCHRITT**

- such im Teilgraphen **TEILGR** ab einem Startknoten **STKN** eine in den Baum **BAUM** aufzunehmende Kante,
- sucht dabei rekursiv im restlichen Teilgraphen weiter, falls dessen erste Kante nicht aufgenommen werden kann (etwa weil sie schon enthalten ist) bzw.
- greift auf die im Stack **STACK** verwalteten Lösungsvarianten zurück, falls keine im Teilgraphen enthaltene Kante aufgenommen werden kann:

```
(DEFUN SUCHSCHRITT (GR TEILGR STKN STACK BAUM)
 (LET ((RESTGR (NEUE_KANTE TEILGR STKN BAUM)))
      (COND ((NULL RESTGR)
                (COND ((NULL STACK) BAUM)
                      ( T (SUCHSCHRITT
                              GR (CDAR STACK) (CADAAR STACK)
                              (CDR STACK) BAUM))))
            ( T (SUCHSCHRITT
                    GR GR (CADAR RESTGR) (CONS RESTGR STACK)
                    (CONS (CAR RESTGR) BAUM))))))
```

Initiale Aufrufbelegung von SUCHSCHRITT

- **GR** = gegebener Graph
- **TEILGR** = gegebener Graph
- **STKN** = Wurzelknoten des Erreichbarkeitsbaumes
- **STACK** = **NIL**
- **BAUM** = **‘((ANF WURZ))**

Die Funktion **ERREICHBARKEITSBAUM** ruft **SUCHSCHRITT** mit diesen Parametern auf

```
(DEFUN ERREICHBARKEITSBAUM (GR WRZ)
   (SUCHSCHRITT GR GR WRZ NIL ‘((ANF WURZ))))
```

Programmaufruf zur Berechnung des Erreichbarkeitsbaumes

```
➢ (SETQ BSPGRAPH ((2 3)(1 5)(3 1)(1 4)(5 6)(4 2)(6 5)(6 3)(6 4)))
  ((2 3)(1 5)(3 1)(1 4)(5 6)(4 2)(6 5)(6 3)(6 4)))
➢ (ERREICHBARKEITSBAUM BSPGRAPH 1)
  ((4 2)(6 4)(6 3)(5 6)(1 5)(ANF 1))
```

```
(DEFUN NAECHSTE_KANTE (GRAPH STARTKNOTEN)
    (COND        ((NULL GRAPH)                      NIL)
                 ((EQUAL STARTKNOTEN (CAAR GRAPH)) GRAPH)
                 ( T  (NAECHSTE_KANTE (CDR GRAPH STARTKNOTEN))))
(DEFUN ENTHALTEN (KNOTEN BAUM)
    (COND        ((NULL BAUM)                              NIL)
                 ((EQUAL KNOTEN (CADAR BAUM))      T)
                 ( T            (ENTHALTEN KNOTEN (CDR BAUM)))))
(DEFUN NEUE_KANTE (GR STKN BAUM)
 (COND ((NULL GR)NIL)
       ( T (LET ((RESTGR (NAECHSTE_KANTE GR STKN)))
              (COND ((ENTHALTEN (CADAR RESTGR) BAUM)
                                    (NEUE_KANTE (CDR RESTGR) STKN BAUM))
                    ( T      RESTGR))))))
(DEFUN SUCHSCHRITT (GR TEILGR STKN STACK BAUM)
 (LET ((RESTGR (NEUE_KANTE TEILGR STKN BAUM)))
      (COND ((NULL RESTGR)
                  (COND ((NULL STACK) BAUM)
                        ( T (SUCHSCHRITT GR (CDAR STACK)
                            (CADAAR STACK) (CDR STACK) BAUM))))
            ( T (SUCHSCHRITT GR GR (CADAR RESTGR)
                  (CONS RESTGR STACK)(CONS (CAR RESTGR) BAUM))))))
(DEFUN ERREICHBARKEITSBAUM (GR WRZ)(SUCHSCHRITT GR GR WRZ NIL '((ANF WURZ))))
```

Zum Vergleich: Eine PROLOG-Lösung, welche die gleiche Strategie umsetzt

```
naechste_kante( [[S,Z]|RestGr] , S , [[S,Z]|RestGr] ) :- !.
naechste_kante( [_|RestGr] , S , Gr ) :- naechste_kante( RestGr , S , Gr).

enthalten( K , [[_ , K]|_] ) :- !.
enthalten( K , [_|R] ) :- enhalten( K , R ).

neue_kante(Gr, S, Baum, NeueKante) :-
        naechste_kante(G, S, RestGr), knotenausw(RestGr, S, Baum, NeueKante).

knoten_ausw( [[S,Z]|Rest], S, Baum, NeueKante) :-
        enthalten(Z, Baum), !, neue_kante(Rest, S, Baum,NeueKante).
knoten_ausw(RestGr,_,_,_,RestGr).

suchschritt(Gr, TeilGr, S, Stack, Baum, NBaum) :-
        neue_kante(TeilGr, S, Baum, RestGr),
        restgr_ausw(RestGr, Gr, TeilGr, S, Stack, Baum, NBaum).

restgr_ausw( [ ], _, _, _, [ ], Baum, Baum) :- !.
restgr_ausw( [ ], Gr, TeilGr, S, [[[S,Z]|R]|Rest], Baum, NBaum) :- !,
        suchschritt(Gr, R, Z, Rest, Baum, NBaum).
restgr_ausw( [[S,Z]|Rest], Gr, TeilGr, S, Stack, Baum, NBaum) :-
        suchschritt(Gr, Gr, Z, [[[S,Z]|Rest]|Stack], [[S,Z]|Baum], NBaum).

erreichbarkeitsbaum(Gr, Wurz, Baum) :- suchschritt(Gr, Gr, Wurz, [ ], [[anf,Wurz]], Baum).
```