

SE III: Logikprogrammierung

Wolfgang Menzel

Telefon: 42883-2435

E-Mail: menzel@informatik.uni-hamburg.de

Raum: F-411

Sprechstunde: Di 16 - 17 Uhr

Der Zyklus “Softwareentwicklung”

Vier Grundbausteine

- SE1/SE2: Zustands- und Objektorientierte Programmierung
- SE3: Programmieren mit Funktionen / Relationen
- ein Praktikum (IP11)
 - Softwaretechnik, (Logikprogrammierung,) Bildverarbeitung, Internetwerkzeuge, Roboterprogrammierung, IT-Sicherheit . . .

Lernziele

- Kennenlernen von grundlegenden Konzepten, Methoden und Werkzeugen der Softwareentwicklung in einem alternativen Programmierparadigma
- Erwerben der Fähigkeit zur kritischen Auswahl von Methoden und Werkzeugen
- Erwerben der Fähigkeit zum selbständigen und systematischen Lösen programmiertechnischer Aufgaben
- Vertiefung von formalen Konzepten der Informatik anhand von konkreten Programmierproblemen

Lernformen

- Die Vorlesung:
Konzepte, Begriffe, Zusammenhänge, Beispiele
- Das Selbststudium:
Vertiefung, Literaturstudium, Klärung technischer Details,
praktische Programmierübungen, Lerngruppenarbeit
- Die Übung:
Hausaufgaben, Präsenzübungen, Klären von
Verständnisproblemen
- Die Rückkopplung:
Anregungen, Kritik

Die Übungen

Betreuer	Raum	Tag	Zeit	Anmeldungen
Wolfgang Menzel	F-534	Di	8-10	100
Marco Form	G-102	Mi	8-10	86
Michael König	F-534	Mi	10-12	93
Colin Maier	C-104	Mi	12-14	106
Timo Baumann	G-210	Mi	12-14	106
Marco Form	G-203	Mi	14-16	104
Timo Baumann	G-210	Mi	14-16	102
Michael König	G-210	Do	8-10	103
Michael König	G-210	Do	10-12	108

Die Übungen

- Erste Übung: 25./26./27.10.2016
- Ausgabe der Übungsaufgaben im NatsWiki
Department Informatik → Einrichtungen → Arbeitsbereiche → NATS → Courses: SE3 Logikprogrammierung
- Bearbeitung der Aufgaben überwiegend am Rechner (unbetreut)
- Diskussion der Lösungen in der Übungsgruppe
- Abgabe der Lösungen
 - per Mail bei der/dem Übungsgruppenleiter(in)
 - mit Angabe derjenigen Aufgaben, für die die Bereitschaft zum Vortragen der Lösung besteht
- Bearbeitung in Gruppen (max. 3 Studierende) ist möglich und erwünscht

Die Übungen

- Erfolgreiche Teilnahme: (nachgewiesen durch einen Übungsschein)
 - regelmäßige aktive Teilnahme (85%)
 - mindestens 50% der möglichen Punkte erreicht
 - Punktzahl pro Aufgabenblatt variiert:
10, 15, 20, 25, 30, 30, 30, ...
 - maximale Gesamtpunktzahl: 310
 - mindestens zwei Präsentationen von Lösungsvorschlägen vor der Übungsgruppe

Die Software

- SWI-Prolog
 - Open-Source-Projekt (Universität Amsterdam)
 - Download: www.swi-prolog.org
 - integriertes Objektsystem
 - ausreichend für die Lehrveranstaltung SE III
- Sicstus-Prolog (Swedish Institute of CS, Kista)
 - semi-professionelles System
 - Studentenlizenzen über Michael König (RZ)
 - zahlreiche spezielle Erweiterungen (Coroutinen, Constraints, Constraint-Handling Rules, ...)
 - empfehlenswert für Prolog-Interessenten mit Ambitionen

- Clocksin, W. F. und Mellish, C. S. (2003) Programming in Prolog. 5th edition, Springer-Verlag, Berlin.
- Sterling, L. und Shapiro, E. (1994) The Art of Prolog. Advanced Programming Techniques. 2nd edition, MIT-Press, Cambridge MA.
- Clocksin, W. F. (1997) Clause and Effect. Prolog Programming for the Working Programmer. Springer-Verlag, Berlin.
- Bratko, I. (2012) Prolog - Programming for Artificial Intelligence. 4th edition, Addison-Wesley/Pearson Education, Harlow.

Logikprogrammierung

2. Programmierparadigmen
3. Relationale Datenbanken
4. Deduktive Datenbanken
5. Rekursive Datenstrukturen
6. Verkettete Listen
7. Extra- und Metalogische Prädikate
8. Prädikate höherer Ordnung
9. Funktionale Programmierung
10. Aktive Datenstrukturen
11. Fazit und Ausblick

Programmierparadigmen

```
function member(Element,Liste,Laenge);  
  declare Element,Laenge,I integer;  
  declare Liste(Laenge) array of integer;  
  for I=1 to Laenge do;  
    if Element=Liste(I) then return TRUE;  
  end do;  
  return FALSE;  
end function;
```

```
(define member (element liste)  
  (cond ((null liste) #f)  
        ((equal (car liste) element) #t)  
        (member element (cdr liste)) ) )
```

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

2. Programmierparadigmen

Verarbeitungsmodelle

Abstraktion

Programmierstile

Logikprogrammierung

Verarbeitungsmodelle

berechnungsuniverselle Verarbeitungsmodelle

- imperatives Modell
- funktionales Modell
- logik-basiertes Modell
- ...

eingeschränkte Verarbeitungsmodelle

- relationale Datenbanken
- Tabellenkalkulation
- ...

Imperative Verarbeitung

- Anweisungsfolgen verändern den internen Zustand des Automaten bzw. veranlassen Ein-/Ausgabeoperationen.

→ SE I

```
function member(Element,Liste,Laenge);  
  declare Element,Laenge,I integer;  
  declare Liste(Laenge) array of integer;  
  for I=1 to Laenge do;  
    if Element=Liste(I) then return TRUE;  
  end do;  
  return FALSE;  
end function;
```

Funktionale Verarbeitung

- Durch Auswertung funktionaler Ausdrücke werden Werte (Berechnungsergebnisse) ermittelt.

→ SE III/F

```
(define member (Element Liste)
  (cond ((null Liste) #f)
        ((equal Element (car Liste)) #t)
        (member Element (cdr Liste)) ) )
```

Logik-basierte Verarbeitung

- Durch logische Deduktion wird die Zugehörigkeit von Wertekombinationen zu einer Relation geprüft. Dabei werden Wertebindungen hergestellt, die sich als Berechnungsergebnisse interpretieren lassen.

→ SE III/L

```
member(Element, [Element|_]).  
member(Element, [_|Restliste]) :-  
    member(Element, Restliste).
```

Verarbeitungsmodelle

Weitere berechnungsuniverselle Verarbeitungsmodelle:

objektorientiertes Modell

Botschaften werden an Objekte verschickt und verändern deren internen Zustand. Objekte können durch Abstraktion hierarchisch strukturiert werden. Zustandsänderungen werden imperativ oder funktional formuliert.

→ SE I

constraint-basiertes Modell

Die möglichen Werte von Variablen werden durch die Angabe von Bedingungen sukzessive eingeschränkt (Beschränkungserfüllung).

→ GWV

Verarbeitungsmodelle

Eingeschränkte Verarbeitungsmodelle

- Endliche Automaten, Reguläre Grammatiken, Reguläre Ausdrücke
- Tabellenkalkulationsprogramme
- Markup-Sprachen (HTML, XML, ...)
- Relationale Datenbanken

→ FGI I

→ GDB

weitere Verarbeitungsmodelle

→ STOYAN (1988) Programmiermethoden der Künstlichen Intelligenz

Abstraktion

- Alle Verarbeitungsmodelle sind Abstraktionen von der zugrundeliegenden Hardware → "Abstrakte Maschine"

 elementare Operationen des Prozessors → (komplexe) Operationen der abstrakten Maschine

 (virtueller) Speicher → abstraktes Speichermodell
- Befreiung der Programmierung vom VON NEUMANN-Stil

Abstraktionsniveau

- Verarbeitungsmodelle abstrahieren unterschiedlich stark von der zugrundeliegenden Hardwarestruktur

	imperativ	funktional	logik-basiert
elementare Datentypen	–	–	(–)
Anweisungslogik	+	++	+++
Steuerstrukturen	+	++	+++
Speichermodell	–	++	+++
Verarbeitungsrichtung	–	–	++

Abstraktion

Abstraktion ...

- ... ist das Hervorheben bestimmter Merkmale eines Objekts als relevant
- ... ist das Vernachlässigen anderer Merkmale eines Objekts als irrelevant

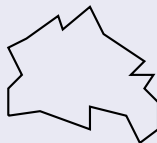
Abstraktion ...

- ... ist ein wesentliches Hilfsmittel zur Reduktion von Komplexität
- ... ist eine zentrale Kulturleistung des Menschen
- ... ist aufgabenbezogen und subjektiv
- ... kann daher eine wesentliche Quelle für Missverständnisse sein

Abstraktion

Abstraktion reduziert Komplexität

Beispiel: Entfernung zwischen zwei Städten



Abstraktion als Kulturleistung

Beispiel: Handel

Ware gegen Ware

→ Ware gegen Materialwert

→ Ware gegen Nominalwert

→ Ware gegen Verrechnungseinheit

Abstraktion als Quelle von Missverständnissen

Beispiel: Maßeinheiten

°C oder °F ?

km oder Meilen?

km/h oder mph?

Welche Meilen?

Abstraktion in der Softwareentwicklung

Abstraktion vom Einzelfall	→	Wiederverwendbarkeit
Abstraktion von irrelevanten Aspekten	→	Anwendungsbezug
Abstraktion von der Hardware	→	Komplexitätsreduktion

Abstraktion durch Generalisierung

- Suche nach regelhaften Zusammenhängen in den Daten
- Wiederverwendbarkeit für große Klassen von Beispielfällen

funktionale Abstraktion

$x = 1, f(x) = 1$

$x = 2, f(x) = 4$

$x = 3, f(x) = 9$

```
(define square(x) (* x x))
```

relationale Abstraktion

`vogel(amsel) kann_fliegen(amsel)`

`vogel(star) kann_fliegen(star)`

`vogel(meise) kann_fliegen(meise)`

```
kann_fliegen(X) :- vogel(X).
```

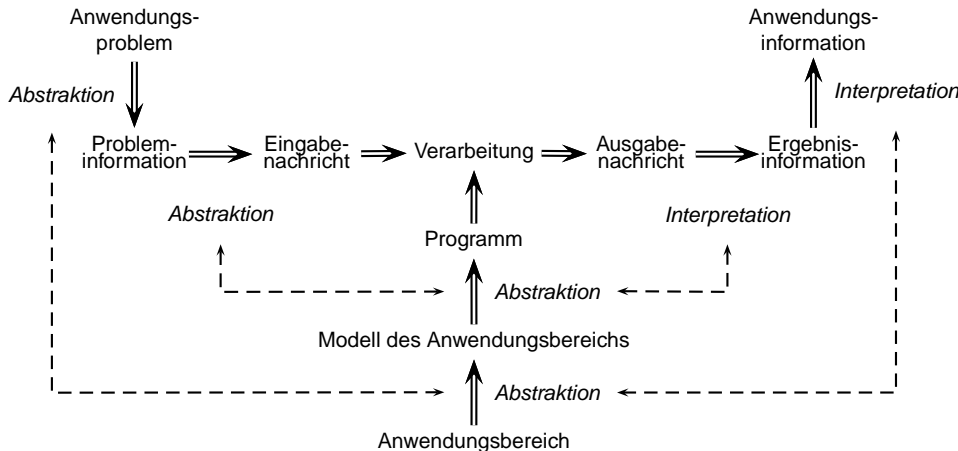

Abstraktion durch Vergröberung

- Weglassen von Sonder- und Ausnahmefällen
- Weglassen von Maßeinheiten
- Weglassen von Ausgabeinformation
- Weglassen von Kommentaren

- ... in Abhängigkeit von der Relevanz für eine Aufgabenstellung

- aber: Relevanzbewertungen sind hochgradig subjektiv!

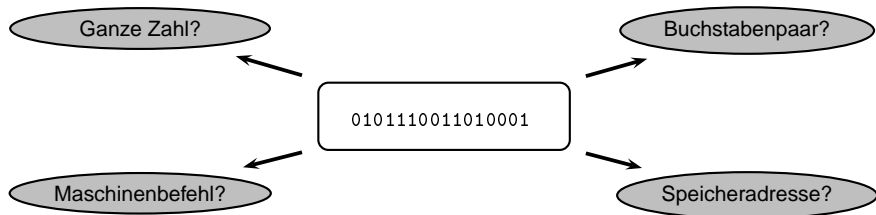
Abstraktion durch Vergrößerung



Abstraktion durch Verbergen

- Abstraktion als Voraussetzung für universell einsetzbare Hardware:

Beispiel: Binärcodierung



Abstraktion durch Verbergen

- Abstraktion von Möglichkeiten der Hardware
→ Voraussetzung für den verlässlichen Umgang mit ihr
- Beispiel: goto considered harmful
 - DIJKSTRA 1968
Jedes Programm, das goto-Befehle enthält, kann in ein goto-freies Programm umgewandelt werden. → SE I
- Beispiel: Wertmanipulation als Fehlerquelle
 - Berechnungsergebnis ist vom Zustand eines Programms abhängig
 - Zustand ist zum Entwicklungszeitpunkt nicht sichtbar
 - Bedeutung eines Programmes ist von (lokal) nicht sichtbaren Eigenschaften abhängig: Seiteneffekte

Abstraktion durch Verbergen

- Umgang mit Seiteneffekten: Unterschiedliche Lösungsansätze
- Objektorientierung:
 - Zustandsänderungen nur über wohldefinierte Schnittstellen (Methoden)
 - Einschränkung auf lokale Gültigkeitsbereiche (Klassendefinition)
- funktionale Programmierung:
 - Zurückdrängen von Wertemanipulationsmöglichkeiten
 - aber nichtlokale Gültigkeitsbereiche
- Logikprogrammierung:
 - Wertemanipulation nur durch das Verarbeitungsmodell
 - extrem lokale Gültigkeitsbereiche

Abstraktion

- ... ist ein wesentlicher Bestandteil der Softwareentwicklung
- ... erlaubt kompakte und überschaubare Problemlösungen
- ... kann leicht zur Fehlerquelle werden
- ... erfordert *Sorgfalt* bei Entwicklung und Dokumentation
 - insbesondere für die als irrelevant betrachteten Aspekte!
 - erfordert das “Hineinversetzen” in den Adressaten!
 - Dokumentation der Entwicklerintention in vielfältigen Formen: Programmkommentare, Handbücher, Gebrauchsanweisungen, ...

Relationale Abstraktion

- Logikprogrammierung abstrahiert auch vom Algorithmusbegriff
- Ein Algorithmus beschreibt einen (parametrisierbaren) dynamischen Ablauf
 - Eingabedaten
 - prozeduraler Ablauf
 - Ausgabedaten
- Logikprogramme beschreiben den generellen Zusammenhang zwischen zwei oder mehr Datenstrukturen und lassen prinzipiell beliebige Verarbeitungsrichtungen zu

Relationale Abstraktion

- Kontrast: funktionale Programmierung ist immer richtungsabhängig

```
(define member (Element Liste)
  (cond ((null Liste) #f)
        ((equal Element (car Liste)) #t)
        (member Element (cdr Liste)) ) )
```

- einzig mögliche Abbildung: $\text{Element} \times \text{Liste} \mapsto \{\#t, \#f\}$

```
(member 'a '()) ==> #f
(member 'b '(a b c)) ==> #t
(member 'd '(a b c)) ==> #f
```


Relationale Abstraktion

- Ein Logikprogramm kann üblicherweise unterschiedliche algorithmische Abläufe realisieren:

```
member(Element, [Element|_]).  
member(Element, [_|Restliste]) :-  
    member(Element, Restliste).
```

- Aufrufvariante 1: $\text{Element} \times \text{Liste} \mapsto \{\text{true}, \text{false}\}$

```
?- member(a, []).  
false.  
?- member(b, [a,b,c]).  
true.  
?- member(d, [a,b,c]).  
false.
```

Relationale Abstraktion

- Aufrufvariante 2: Liste \mapsto Element

```
?- member(E, []).  
false.  
?- member(E, [a,b,c]).  
E = a ;  
E = b ;  
E = c .
```

Relationale Abstraktion

- Aufrufvariante 3: $\text{Element} \mapsto \text{Liste}$

```
?- member(a,L).
```

```
L = [a|_] ;
```

```
L = [_,a|_] ;
```

```
L = [_,_,a|_] ;
```

```
L = [_,_,_,a|_] .
```

- Aufrufvariante 4: $\emptyset \mapsto \text{Element} \times \text{Liste}$

Relationale Abstraktion

- Logikprogramme implementieren nicht konkrete Algorithmen sondern generalisierte Algorithmenschemata
- Logikprogramme kodieren nicht das Lösungsverfahren sondern das zur Problemlösung erforderliche (statische) Wissen über die Zusammenhänge zwischen den formalen Parametern

Sagen, wie es ist.

KAY V. LUCK

- Logikprogramme sind im allgemeinen Fall *richtungsunabhängig*.

Relationale Abstraktion

- die algorithmischen Details müssen nicht im Programm ausgedrückt werden sondern sind bereits Bestandteil des Verarbeitungsmodells

algorithm = logic + control

(BOB KOWALSKI)

Programm

abstrakte Maschine

- die Verarbeitungsrichtung wird erst beim Aufruf festgelegt

Hybride Modelle

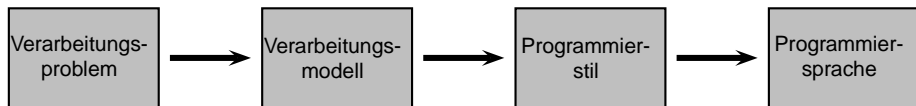
Verarbeitungsmodelle können miteinander kombiniert werden:

- imperativ + Elemente von funktional (C, Java)
- imperativ + objektorientiert (C++)
- funktional + objektorientiert (Common Lisp)
- logik-basiert + Elemente von funktional (Prolog)
- funktional + relational
- logik-basiert + constraint-basiert (CLP)
- ...

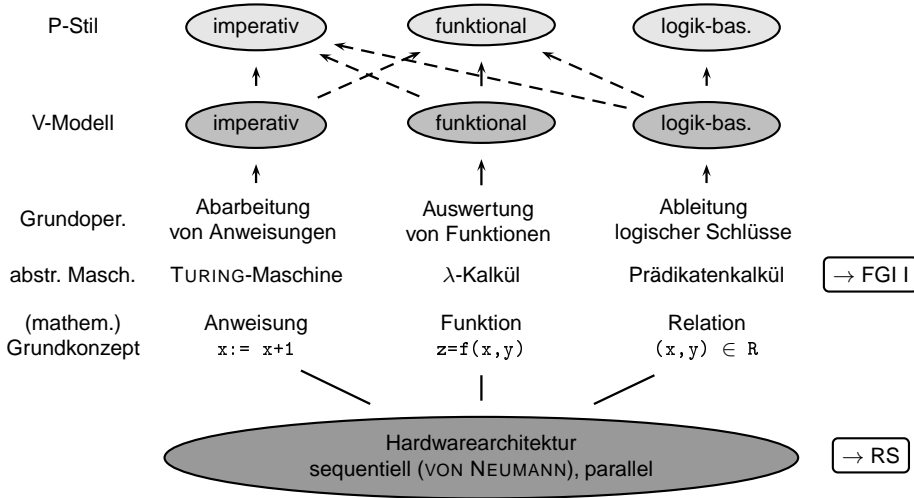
Programmierstil

Ein Verarbeitungsmodell begründet einen bestimmten Programmierstil, ist aber nicht auf diesen eingeschränkt.

- Jedes Verarbeitungsproblem legt ein geeignetes Verarbeitungsmodell nahe.
- Verarbeitungsmodelle sind Denkwerkzeuge.
- Ein bestimmtes Verarbeitungsmodell wird durch die einzelnen Programmiersprachen mehr oder weniger gut unterstützt.
- Jede Programmiersprache fördert einen bestimmten Programmierstil besonders.



Verarbeitungsmodell - Programmierstil



Logikprogrammierung

- hohes Abstraktionsniveau
- sehr mächtige Werkzeuge der Informatik als Basiskonstrukte
 - Datenbanken, nichtdeterministische Suche, Mustervergleich, ...
- unmittelbarer Bezug zu zentralen Konzepten der theoretischen Informatik und Mathematik
 - Relation, Funktion, Unifikation, Substitution, Subsumtion, Verband, Unterspezifikation, Transducer, Grammatik, Symmetrie, Transitivität, Ableitung, ...
- kompakte Problemlösungen
- direkter Bezug zur Problemstellung
- schnelle Realisierung von funktionsfähigen Prototypen

Logikprogrammierung

- berechnungsuniverselle Programmiersprache durch Kombination weniger, aber sehr mächtiger Basiskonzepte
 - Unifikation von rekursiven (Daten-)Strukturen
 - Termersetzung und Variablensubstitution
 - nichtdeterministische Suche
- Syntax und Semantik der Sprache können leicht modifiziert werden
 - Anpassung an spezielle Nutzungsbedingungen
 - experimentelles Prototyping für neue Verarbeitungsmodelle

Name	Vorname	Gehalt
Meier	Kerstin	3400
Schulze	Hans	5300
Müller	Annegret	6300
Heimann	Manfred	2500

3. Relationale Datenbanken

Relationen

Fakten

Anfragen

Anfragen mit Variablen

Komplexe Anfragen

Prädikate zweiter Ordnung

Relationale DB-Systeme

Relationen

Relation in M

Teilmenge des Kreuzproduktes der Menge M mit sich selbst.

$$R^{(2)} \subseteq M \times M = M^2$$

Kreuzprodukt (Cartesisches Produkt) zweier Mengen A und B

Menge aller geordneten Paare (2-Tupel), die sich aus den Elementen von A und B bilden lassen:

$$A \times B = \{(x, y) | x \in A, y \in B\}$$

Relationen

Domäne einer Relation

Menge M , über der die Relation definiert ist.

Extensionale Spezifikation einer Relation

Aufzählung ihrer Elemente

Beispiele für Relationen

$$A = \{o, O, \bigcirc\}$$

$$A \times A = \left\{ \begin{array}{lll} (o, o), & (O, o), & (\bigcirc, o), \\ (o, O), & (O, O), & (\bigcirc, O), \\ (o, \bigcirc), & (O, \bigcirc), & (\bigcirc, \bigcirc) \end{array} \right\}$$

$$\text{"ist gleich groß" (in } A) = \{(o, o), (O, O), (\bigcirc, \bigcirc)\} \subseteq A^2$$

$$\text{"ist kleiner als" (in } A) = \{(o, O), (o, \bigcirc), (O, \bigcirc)\} \subseteq A^2$$

$$\begin{aligned} \text{"ist größer oder gleich" (in } A) &= \{(o, o), (O, o), (\bigcirc, o), \\ &\quad (o, O), (O, O), (\bigcirc, O)\} \subseteq A^2 \end{aligned}$$

Relationen

sind Beziehungen zwischen Objekten, Ereignissen und abstrakten Ideen und Begriffen.

- Relation ist ein formales Konstrukt
- Problem bei der Anwendung auf "reale" Fragestellungen
 - Objekte und Begriffe treten niemals isoliert auf.
 - vollständige Erfassung (Verstehen) eines Begriffs erst durch Berücksichtigung der Gesamtheit seiner Relationen zu anderen Objekten und Begriffen möglich.
- Approximation nötig:
 - *Weitgehende* Erfassung ...
 - Berücksichtigung der *wesentlichen* Relationen ...
 - → Abstraktion durch Vergrößerung

Modellierung mit Relationen

- Relationen im strengen Sinne: zweistellige Relationen
 - A "ist kleiner als" B
 - A "passierte zeitlich nach" B
 - A "ist Voraussetzung für" B
- Allgemeiner Fall: n-stellige Relationen ($n \geq 1$)
 - A "liegt zwischen" B "und" C
 - A "transportiert" B "von" C "nach" D
- Spezialfall: einstellige Relationen (Eigenschaften, Prädikate)
 - A "ist groß"
 - A "ist blau"
 - A "ist ein Haus"

Modellierung mit Relationen

Beispiel: zweistellige Relation

$$P = \{\text{Anna}, \text{Susi}, \text{Elli}\}$$

$$\begin{aligned} \text{"ist Mutter von" (in } P) &= \{(\text{Susi}, \text{Anna}), \\ &\quad (\text{Elli}, \text{Susi})\} \\ &\subseteq P^2 \end{aligned}$$

Beschränkung auf *eine* Domäne ist zu restriktiv für viele praktische Anwendungen

Erweiterungen

- Erster Schritt:

Verallgemeinerung auf unterschiedliche Domänen

$$R^{(2)} \subseteq M \times N, \quad M \neq N$$

$$B = \{\circ, \square, \triangle\}, \quad C = \{\bullet, \blacksquare, \blacktriangle\}$$

$$\text{"ist kongruent" (über } B \text{ und } C) = \{(\circ, \bullet), (\square, \blacksquare), (\triangle, \blacktriangle)\} \subseteq B \times C$$

$$P = \{\text{Anna, Susi, Elli}\}, \quad N = \{0, 1, 2, \dots\}$$

$$\begin{aligned} \text{"hat_alter" (über } P \text{ und } N) &= \{(\text{Anna}, 12), (\text{Susi}, 35), (\text{Elli}, 63)\} \\ &\subseteq P \times N \end{aligned}$$

Erweiterungen

- Zweiter Schritt:
Verallgemeinerung auf n-stellige Relationen

$$R^{(n)} \subseteq \underbrace{A \times \dots \times Z}_{n \text{ Faktoren}}$$

$$P = \{\text{Anna, Susi, Elli}\}, \quad N = \{0, 1, 2, \dots\}$$

$$F = \{\text{ledig, verheiratet, geschieden, verwitwet}\}$$

$$\begin{aligned} \text{"hat_alter_und_familienstand"} & \text{ (über } P, N \text{ und } F) \\ &= \{(\text{Anna}, 12, \text{ledig}), \\ & \quad (\text{Susi}, 35, \text{geschieden}), \\ & \quad (\text{Elli}, 63, \text{verheiratet})\} \\ &\subseteq P \times N \times F \end{aligned}$$

Relationale Datenbank

Aufzählung von Faktenwissen über einen zu modellierenden Gegenstandsbereich (extensionale Definition).

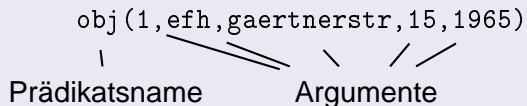
id	type	str	no	jahr
1	EFH	Gaertnerstr	15	1965
2	EFH	Bahnhofsstr	27	1943
3	EFH	Bahnhofsstr	29	1955
4	MFH	Bahnhofsstr	28	1991
5	Bahnhof	Bahnhofsstr	30	1901
6	Kaufhaus	Bahnhofsstr	26	1997
7	EFH	Gaertnerstr	17	1982

Fakten

Fakten

elementare Klauseln, die genau eine *Grundstruktur* enthalten

Struktur



Grundstruktur

Struktur, deren Argumente atomar sind.

Fakten

- Syntax:

Klausel	::=	(Fakt ...) ''
Fakt	::=	Struktur
Struktur	::=	Name ['(' Term {',' Term} ')']
Term	::=	Konstante ...
Konstante	::=	Zahl Name gequoteter_Name
Name	::=	Kleinbuchstabe {alphanumerisches_Symbol}
- Semantik: Jedes Faktum spezifiziert ein Element einer Relation mit dem angegebenen Prädikatsnamen und der Stelligkeit der Struktur.
 - Stelligkeit der Struktur ist Bestandteil der Prädikatsdefinition.
 - Strukturen unterschiedlicher Stelligkeit definieren unterschiedliche Relationen.

- Pragmatik: Eindeutige Prädikatsidentifizierung erfordert die Angabe der Stelligkeit:

`obj/5, liebt/2, teil_von/2 ...`

`singt/1, singt/2 ...`

Prädikat (Relation, Prozedur)

Menge von Fakten mit gleichem Namen und gleicher Stelligkeit

```
obj(1,efh,gaertnerstr,15,1965).  
obj(2,efh,bahnhofsstr,27,1943).  
obj(3,efh,bahnhofsstr,29,1955).  
obj(4,mfh,bahnhofsstr,28,1991).  
obj(5,bahnhof,bahnhofsstr,30,1901).  
obj(6,kaufhaus,bahnhofsstr,26,1997).  
obj(7,efh,gaertnerstr,17,1982).
```


Prädikate

Prädikatsschema

Interpretationshilfe für die Argumentstellen eines Prädikats

```
% obj (Objekt-Nr,Objekttyp,Strasse,Hausnummer,Baujahr)
```

Dient nur der zwischenmenschlichen Kommunikation

Datenbasis

Menge von Prädikatssdefinitionen.

Wird aus einem File eingelesen (`consult/1`).

Anfrage (Ziel, Goal)

elementare Klausel (am Systemprompt eingegeben)

```
?- obj(1,efh,gaertnerstr,15,1965).
```

- Syntax:
Klausel ::= (Fakt | Ziel | ...) ''
Ziel ::= elementares_Ziel | ...
elementares_Ziel ::= Struktur
- Semantik: Prüfen auf Konsistenz mit den Fakten der Datenbasis
 - Ableitbarkeit, Beweisbarkeit bezüglich einer Axiomenmenge (Datenbasis)
 - abgeschlossene Welt (closed world assumption), vollständige Beschreibung

Denotationelle Semantik

Implementationsunabhängige Beschreibung der Bedeutung, wobei das betreffende sprachliche Konstrukt als statisches Objekt betrachtet wird.

Operationale Semantik

Beschreibung der dynamischen Aspekte eines Programms, des durch das Programm erzeugten Verhaltens.

Denotationelle Semantik eines Ziels

Folgt das Ziel (eine logische Aussage) aus der Datenbasis (Axiomenmenge)?

```
?- obj(1,efh,gaertnerstr,15,1965).  
true.  
?- obj(1,efh,gaertnerstr,15,1966).  
false.
```

Negatives Resultat bezieht sich nur auf die gegebene Datenbasis (abgeschlossene Welt)

Operationale Semantik eines Ziels:

Suche nach einem *unifizierbaren* Axiom in der Datenbasis.

- Suche: Systematisches Durchmustern von Entscheidungsalternativen.
- Suchstrategie: Festlegungen über die Reihenfolge bei der Betrachtung der Alternativen.
- Prolog: Durchsuchen der Datenbasis von oben nach unten.
- Negatives Resultat: Erfolglose Suche

Unifikation (I)

Unifikation

Überprüft die Verträglichkeit zweier Strukturen

- Semantik: Unifikation von Grundstrukturen ist ein Identitätstest.
- Prolog: Vergleich des Ziels mit den Fakten der Datenbasis
- Vergleichskriterien:
 - identischer Prädikatsname
 - gleiche Stelligkeit
 - paarweise identische Argumente

Beispieldatenbasis 1

```
% liebt(Wer,Wen-oder-was)
% Wer und Wen-oder-was sind Namen, so dass Wen-oder-was von
% Wer geliebt wird
liebt(hans,geld).
liebt(hans,susi).
liebt(susi,buch).
liebt(karl,buch).
```

Kommentare

Jede Prädikatsdefinition sollte zumindest mit

1. dem Prädikatsschema und
2. *Zusicherungen* über die zulässigen Argumentbelegungen kommentiert werden.

Anfrageabarbeitung

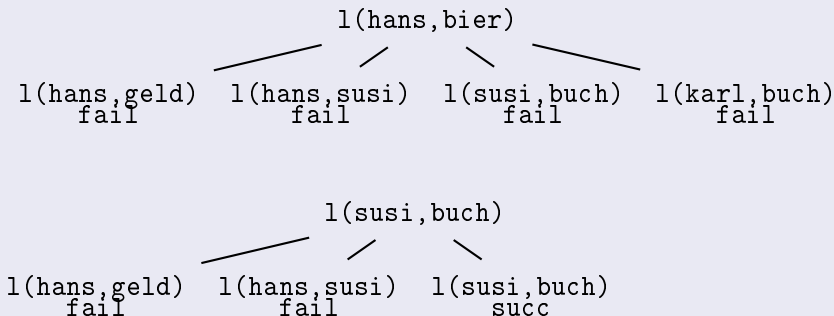
- Operationale Semantik: Veranschaulichung durch ein Ablaufprotokoll (Trace)

```
?- liebt(hans,bier).  
    liebt(hans,geld).      fail  
    liebt(hans,susi).      fail  
    liebt(susi,buch).      fail  
    liebt(karl,buch).      fail  
false.
```

```
?- liebt(susi,buch).  
    liebt(hans,geld).      fail  
    liebt(hans,susi).      fail  
    liebt(susi,buch).      succ  
true.
```

Anfrageabarbeitung

- Operationale Semantik:
Darstellung des Suchraumes als Baum
 - Mutterknoten: Anfrage,
 - Tochterknoten: Fakten aus der Datenbasis



Suchreihenfolge

In Prolog erfolgt die Suche in der Datenbasis in der Reihenfolge der Einträge (d.h. von oben nach unten).

Beobachtung

Für die angegebenen Beispiele sind die denotationelle und die operationale Semantik identisch.

Bezugstransparenz

Eine Sprache, bei der die denotationelle und die operationale Semantik zusammenfallen, nennt man referentiell transparent bzw. deklarativ.

Referentielle Transparenz bedeutet insbesondere, dass das Berechnungsergebnis vom Zustand der (abstrakten) Maschine unabhängig ist.

Anfragen mit Variablen

- universelle Programmiersprachen spezifizieren
Berechnungsvorschriften
 - gewünschtes Produkt: Berechnungsergebnisse (Datenobjekte)
 - bisher: Berechnungsergebnisse auf BOOLE'sche Werte eingeschränkt (Zugehörigkeit zur Relation)
- Erweiterung:
Ermittlung von Berechnungsergebnissen über Variable:
 - Datenobjekte werden an Bezeichner gebunden
 - Variablenbindung (-belegung, -substitution) wird als Berechnungsergebnis ausgegeben

Anfragen mit Variablen

Variable

Paar aus Bezeichner und gebundenem Objekt (z.B. Name, numerischer Wert)

Variablenbindung

Einem Variablenbezeichner wird ein Datenobjekt eindeutig zugeordnet. Die Variable wird instanziiert.

Instanziierungsvarianten

- nichtinstanziierte Variable: Identität des Objekts noch unbekannt
- instanziierte Variable: Bindung an ein Objekt hergestellt

Variable

- Syntax:

Term	::=	Konstante Variable ...
Variable	::=	benannte_Variable ...
benannte_Variable	::=	Großbuchstabe {alphanumerisches_Symbol}

Gültigkeitsbereich (Sichtbarkeitsbereich)

Der Gültigkeitsbereich einer Variablen ist die Klausel.

- keine Referenz auf einen Speicherplatz!
→ keine Zustandsabhängigkeit!
- keine Blockstruktur, keine "Verschattung"
→ keine Skopusfehler
- interne Umbenennung von Variablen zur Konfliktvermeidung nötig

Anfrage mit Variablen

- Variable in einer Anfrage müssen nicht instantiiert sein

Unterspezifikation

- vollständig spezifiziertes Ziel
`?- liebt(susi,karl).`
- partiell unterspezifiziertes Ziel
`?- liebt(susi,X).`
- vollständig unterspezifiziertes Ziel
`?- liebt(X,Y).`

Anfragen mit Variablen

- Semantik:

Prüfen der Konsistenz mit den Eintragungen der Datenbasis unter Unifikation

- Instanziierung von Variablen durch Unifikation.
- Ermitteln einer oder mehrerer Variablenbindungen, die die Aussage des Ziels wahr machen.

→ Erweiterung des Unifikationsbegriffs erforderlich

Unifikation (II)

Unifikation von variablenhaltigen Strukturen

Ermittlung einer solchen Variablensubstitution (Bindung), die die Gleichheit (Verträglichkeit) der Strukturen (Anfrage und Datenbasiseintrag) herstellt

- Vergleichskriterien:
 - identischer Prädikatsname
 - gleiche Stelligkeit
 - Unifikation der Argumente

Konstante / Konstante	Identitätstest	susi = karl
Variable / Konstante	Instanziierung	X = susi
Konstante / Variable	Instanziierung	karl = Y
Variable / Variable	Koreferenz	X = Y

Unifikation (II)

Anfrage	Fakt	Variablensubstitution
liebt(susi,X)	liebt(susi,buch)	X=buch
liebt(X,buch)	liebt(susi,buch)	X=susi
	liebt(karl,buch)	X=karl

Anfragen mit Variablen

- Denotationelle Semantik: Gibt es eine Substitution für den / die Variablenbezeichner in der Anfrage, so dass die Anfrage wahr wird?
- Operationale Semantik: Suche nach einer konsistenten Variablenbindung

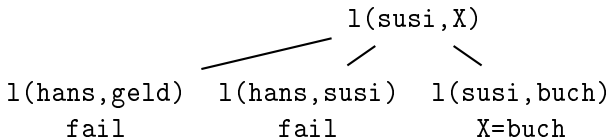
?- `liebt(susi,X).`

`liebt(hans,geld).` `fail`

`liebt(hans,susi).` `fail`

`liebt(susi,buch).` `succ(X=buch)`

`X=buch`



Alternative Variablenbindungen

- Ermittlung alternativer Variablenbindungen: Eingabe von “;” (logisches ODER) im Anschluss an eine Resultatsausgabe

```
?- liebt(susi,X).
```

```
X=buch.
```

```
?- liebt(X,buch).
```

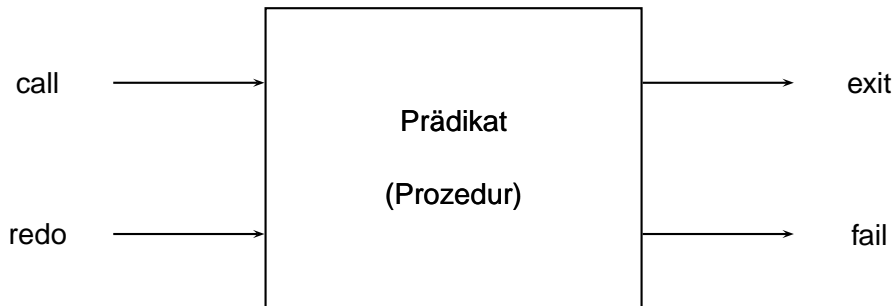
```
X=susi ;
```

```
X=karl.
```

Alternative Variablenbindungen

- Operationale Semantik:
Auffinden alternativer Variablenbindungen durch Backtracking (Zurücksetzen)
 - bei erfolgreicher Unifikation: Zeiger in die Datenbasis setzen
 - bei Aufforderung:
 - Zurückgehen bis zum Zeiger
 - ggf. Variablenbindungen rückgängig machen
 - Suche nach alternativen Fakten in der Datenbasis

Das Vier-Port-Modell



Das Vier-Port-Modell

- ports können als spy-points für `trace/1`, `trace/2`, `debug/0`, `leash/1`, etc. spezifiziert werden

```
?- trace(liebt).
```

```
...
```

```
?- trace(liebt,[call,fail]).
```

```
...
```

```
?- leash(+call).
```

- `leash/1` erlaubt auch 5. "Port": `unify`

Exkurs: Programmiermethodik

Testen eines Prädikats (1)

Beim systematischen Testen eines Prädikats müssen stets

- positive und negative Beispiele und
- alle Instanziierungsvarianten berücksichtigt werden.
- Empfehlenswert ist die Zusammenstellung einer repräsentativen Testmenge bereits vor und während der Programmerarbeitung.

Prädikatsschemata

Im Prädikatsschema werden stets die zulässigen Instanziierungsvarianten vermerkt:

- + nur Eingabeparameter
- Ausgabeparameter
- ? Ein-/Ausgabeparameter

→ SE I

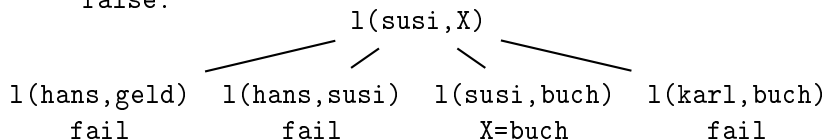
- Beispiel: `liebt(?wer,?wen-oder-was)`

Beispielanfragen

- Anfrage mit einem partiell spezifizierten Ziel (1)

aktuelle Instanziierung: `liebt(+,-)`

```
?- liebt(susi,X).      #1
    liebt(hans,geld).  fail
    liebt(hans,susi).  fail
    liebt(susi,buch).  succ(X=buch)
X=buch ;
    liebt(karl,buch).  fail
false.
```

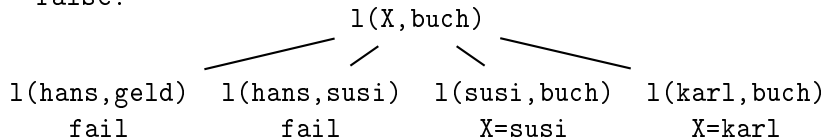


Beispielanfragen

- Anfrage mit einem partiell spezifizierten Ziel (2)

aktuelle Instanziierung: `liebt(-,+)`

```
?- liebt(X,buch).           #1
    liebt(hans,geld).       fail
    liebt(hans,susi).       fail
    liebt(susi,buch).       succ(X=susi)
X=susi ;
    liebt(karl,buch).       succ(X=karl)
X=karl ;
false.
```

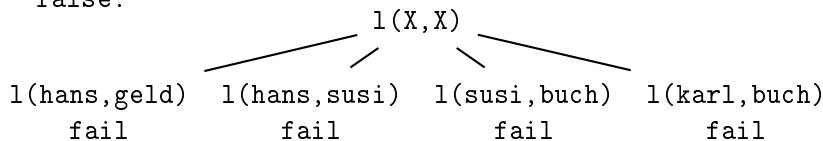


Beispielanfragen

- Anfrage mit einem partiell spezifizierten Ziel (3)

aktuelle Instanziierung: `liebt(-,-)`

```
?- liebt(X,X).           #1
    liebt(hans,geld).    fail
    liebt(hans,susi).    fail
    liebt(susi,buch).    fail
    liebt(karl,buch).    fail
false.
```



Koreferenz (Sharing)

Durch mehrfache Verwendung einer Variablen innerhalb einer Klausel wird *Koreferenz* gefordert.

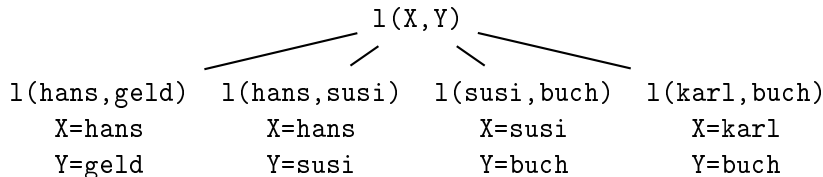
Koreferente Variable haben immer den gleichen Wert (sind an das gleiche Datenobjekt gebunden) sobald sie instanziiert werden.

Koreferenz spezifiziert eine Forderung nach Identität auch wenn der Wert der Variablen zum Aufrufszeitpunkt noch gar nicht bekannt ist.

Beispielanfragen

- Anfrage mit einem vollständig unterspezifizierten Ziel
aktuelle Instanziierung: `liebt(-,-)`

```
?- liebt(X,Y).  
X=hans,Y=geld ;  
X=hans,Y=susi ;  
X=susi,Y=buch ;  
X=karl,Y=buch.
```



Anfragen mit Variablen

Verschieden stark spezifizierte Ziele können unterschiedliches prozedurales Verhalten auslösen:

vollständig spezifiziert	→	Enthaltensein in einer Relation
partiell (unter-)spezifiziert	→	Teilmenge einer Relation
vollständig unterspezifiziert	→	gesamte Relation

Auch bei Einführung von Variablen bleibt die Bezugstransparenz erhalten.

Anfragen mit Variablen

Informationsanreicherung

Relationale Programmierung kann als Prozess der informationellen Anreicherung einer (unterspezifizierten) Anfrage durch die Instanziierung von Variablen interpretiert werden.

`liebt(susi,buch) \Rightarrow { liebt(susi,buch) }`

`liebt(susi,X) \Rightarrow { liebt(susi,buch) }`

`liebt(X,buch) \Rightarrow { liebt(susi,buch), liebt(karl,buch) }`

`liebt(X,X) \Rightarrow \emptyset`

`liebt(X,Y) \Rightarrow { liebt(hans,geld), liebt(hans,susi),
liebt(susi,buch), liebt(karl,buch) }`

Zwischenbilanz: Logikprogrammierung

- Die algorithmische Grundstruktur wird im Verarbeitungssystem festgelegt, nicht im Programm.
- Ein Programm beschreibt eine ganze Klasse von Algorithmen (Algorithmenschema).
- Algorithmische Details, insbesondere die Verarbeitungsrichtung, werden erst durch die Anfrage (Prädikatsaufruf) festgelegt.

Die Richtungsunabhängigkeit (Reversibilität) ist ein wesentliches Merkmal der relationalen Programmierung, für das in den anderen Verarbeitungsmodellen kein Äquivalent existiert: Ein Programm realisiert in Abhängigkeit vom Prädikatsaufruf ganz unterschiedliche prozedurale Abläufe.

Komplexe Anfragen

- Konjunktion mehrerer Teilziele

```
?- liebt(X,susi),liebt(X,geld).  
X=hans.
```

- Syntax:

$$\begin{aligned}\text{Ziel} &::= \text{elementares_Ziel} \mid \text{komplexes_Ziel} \\ \text{komplexes_Ziel} &::= \text{elementares_Ziel} \text{ ',' } \text{Ziel}\end{aligned}$$

- Denotationelle Semantik: logisches UND
- Operationale Semantik: konjunktiv verknüpfte Ziele werden sequentiell abgearbeitet.
Die Konjunktion war erfolgreich, wenn alle Teilziele erfolgreich waren.

Komplexe Anfragen

- Disjunktion mehrerer Teilziele

```
?- liebt(X,susi);liebt(susi,X).
```

```
X=hans ;
```

```
X=buch.
```

- Syntax: komplexes_Ziel ::= elementares_Ziel (',' | ';') Ziel
- Denotationelle Semantik: logisches ODER
- Operationale Semantik: disjunktiv verknüpfte Ziele werden sequentiell abgearbeitet. Die Disjunktion war erfolgreich, wenn ein Teilziel erfolgreich war.
- Pragmatik: Vermeiden Sie (vorerst) die Verwendung der Disjunktion, weil sie Programme teilweise schwer verständlich macht.

Suche bei konjunktiv verknüpften Anfragen

Tiefensuche

Eine (partiell) erfolgreiche Variablenbindung wird an den noch verbleibenden Teilzielen überprüft.

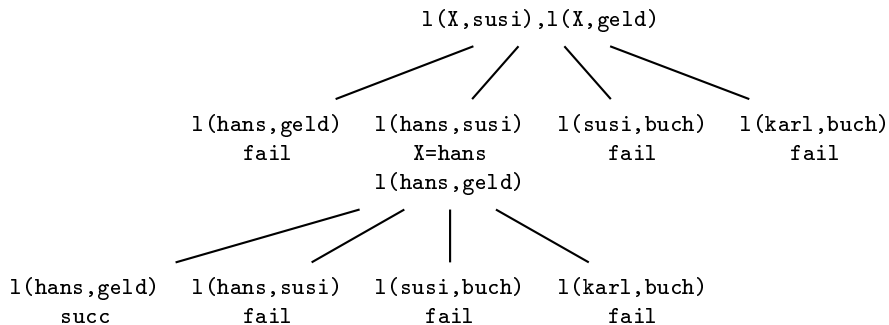
Der Suchbaum wird zuerst in die Tiefe expandiert.

Backtracking:

Wurde ein Teilbaum exhaustiv, aber erfolglos durchsucht, wird zu einem früheren Entscheidungspunkt zurückgegangen.

Beispielanfragen

- konjunktiv verknüpfte Anfrage: `liebt(-,+),liebt(-,+)`
- Suchbaum



Beispielanfragen

- Trace

```
?- liebt(X,susi),liebt(X,geld).  
    ?- liebt(X,susi).                #1  
        liebt(hans,geld).            fail  
        liebt(hans,susi).            succ(X=hans)  
    ?- liebt(hans,geld).             #2  
        liebt(hans,geld).            succ(X=hans)  
X=hans ;  
        liebt(hans,susi).            fail  
        liebt(susi,buch).            fail  
  
        liebt(karl,buch).            fail  
BT #1  
        liebt(susi,buch).            fail  
        liebt(karl,buch).            fail  
false.
```

Anonyme Variable

- Syntax: $\text{Variable} ::= \text{benannte_Variable} \mid \text{anonyme_Variable} \mid \dots$
 $\text{anonyme_Variable} ::= _$
- Semantik: Eine anonyme Variable kann an jeden Wert gebunden werden. Mehrere anonyme Variable innerhalb einer Klausel werden unabhängig voneinander instanziiert (stellen keine Koreferenz her).
- Pragmatik: Anonyme Variable werden verwendet, wenn die Variablenbindung für die jeweilige Berechnungsaufgabe irrelevant ist.
- Beispiel: Welche Mehrfamilienhäuser gibt es?

aktuelle Instanziierung: `obj (_, +, -, -, _)`

```
?- obj (_,mfh,Str,No,_).  
    Str=bahnhofstr, No=28.
```

Anfragen über mehreren Relationen

Beispieldatenbasis 2

```
% obj(?Objektnr,?Objekttyp,?Strassenname,?Hausnr,?Baujahr).  
obj(1,efh,gaertnerstr,15,1965).  
obj(2,efh,bahnhofsstr,27,1943).  
obj(3,efh,bahnhofsstr,29,1955).  
obj(4,mfh,bahnhofsstr,28,1991).  
obj(5,bahnhof,bahnhofsstr,30,1901).  
obj(6,kaufhaus,bahnhofsstr,26,1997).  
obj(7,efh,gaertnerstr,17,1982).
```

```
% bew(?Vorgangsnr,?Objektnr,?Verkaeuer,?Kaeufer,?Preis,  
%      ?Verkaufsdatum).  
bew(1,1,mueller,meier,450000,'1997.01.01').  
bew(2,3,schulze,schneider,560000,'1988.12.13').  
bew(3,3,schneider,mueller,615000,'1996.12.01').  
bew(4,5,bund,piepenbrink,3500000,'2001.06.01').
```


Anfragen über mehreren Relationen

- Beispiel: Wer hat welches Haus gekauft?

```
?- obj(X,_,Str,No,_),bew(_,X,_,Kaeufer,_,_).  
    X=1, Str=gaertnerstr, No=15, Kaeufer=meier ;  
    X=3, Str=bahnhofsstr, No=29, Kaeufer=schneider ;  
    X=3, Str=bahnhofsstr, No=29, Kaeufer=mueller ;  
    X=5, Str=bahnhofsstr, No=30, Kaeufer=piepenbrink.
```

- Der Bezug zwischen den Tabellen wird durch Koreferenz hergestellt.
- Eine komplexe Anfrage mit Variablen definiert eine neue Relation über den betreffenden Domänen.

Prädikate zweiter Ordnung

- ausgewählte Prädikate fordern Prädikatsaufrufe (Ziele) als Eingabewerte
- Beispiel: `findall/3`
 - Syntax: `findall(Term,Ziel,Liste)`
 - Semantik: Aufsammeln *aller* Resultate des Prädikatsaufrufs `Ziel` als instanziierte Varianten des Ausdrucks `Term` in einer Ergebnisliste `Liste`
 - Pragmatik: `Term` und `Ziel` haben üblicherweise gemeinsame (uninstanziierte) Variable.
 - Einzige Instanzierungsvarianten:
`findall(+Term,+Ziel,?Liste)`

Prädikate zweiter Ordnung

- Beispielaufruf

```
?- findall(B,obj(_,efh,_,_,B),L).
```

```
L = [1965, 1943, 1955, 1982].
```

```
?- findall(adresse(Str,Nr),  
           (obj(_,efh,Str,Nr,B),B>1960),  
           L).
```

```
L = [adresse(gaertnerstr, 15), adresse(gaertnerstr, 17)].
```

- weitere Prädikate 2. Ordnung: setof/3, bagof/3

Relationale Datenbank-Systeme

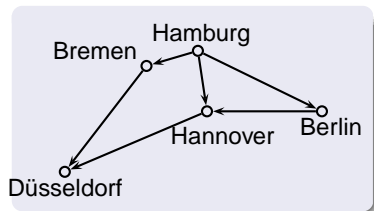
- SQL als Abfragesprache
- Relationenalgebra als Verarbeitungsmodell
 - Selektion: Anfragen
 - Projektion: Anfragen mit anonymen Variablen
 - Join: Mehrtabellenanfragen mit gemeinsamer Variablen
 - aggregierende Operatoren (`count`, `max`, ...): Prädikate zweiter Ordnung

Relationale Datenbankabfragesprachen sind referentiell transparent und richtungsunabhängig.

Relationale Datenbank-Systeme

- der relationale Datenbank-Kern von Prolog ...
 - ist äquivalent zur Recherchefunktionalität von SQL
 - ist eine echte Teilmenge von Prolog
 - ist nicht berechnungsuniversell (wie jede DB-Abfragesprache)
- Datenbanksysteme ...
 - ... ermöglichen persistente Haltung großer Datenbestände
 - ... ermöglichen Mehrnutzerbetrieb
 - ... unterstützen Wartung und Aktualisierung der Daten
 - ... sichern die Integrität der Daten

→ GDB



4. Deduktive Datenbanken

Deduktion

Regeln

Spezielle Relationstypen

Anwendung: Wegplanung

Deduktion

Ableitung von Folgerungen aus einer Axiomenmenge

- Aussagenlogik:
 - Axiome: Atomare Formeln und Implikationen
 - Schlussregel: modus ponens

$$a \wedge (a \rightarrow b) \models b$$

- Prädikatenlogik

$$\frac{\begin{array}{l} \text{Hans ist ein Mann.} \\ \text{Alle Männer sind Verbrecher.} \end{array}}{\text{Hans ist ein Verbrecher.}}$$
$$\frac{\begin{array}{l} \text{mann(hans)} \\ \forall x.\text{mann}(x) \rightarrow \text{verbrecher}(x) \end{array}}{\text{verbrecher(hans)}}$$

Deduktion

- Bisher: Spezifikation einer Relation durch Aufzählung ihrer Elemente (Fakten)
→ extensionale Spezifikation

$$\mathcal{R}_i = \{(x_{11}, \dots, x_{m1}), (x_{12}, \dots, x_{m2}), \dots, (x_{1n}, \dots, x_{mn})\}$$

- Ziel: Berechnung der Elemente einer Relation aufgrund von *Regeln*
→ intensionale Spezifikation

$$\mathcal{R}_i = \{(x_1, \dots, x_n) | \textit{Bedingungen}\}$$

- Regeln entsprechen den Implikationen der Prädikatenlogik
- Ableitung neuer Fakten durch Anwendung von Regeln auf bereits bekannte Fakten

Deduktion

- relationale Abstraktion: Aufbau intensionaler Spezifikationen.
 - Beseitigung von Redundanz
 - Spezifikation von Relationen über unendlichen Domänen

Regeln

- Denotationelle Semantik:
logische Ableitbarkeit mit *modus ponens*
- Operationale Semantik: (Abarbeitung einer Regel)
 - Unifikation des Regelkopfes mit der Anfrage
 - falls erfolgreich:
 - Umbenennen der Variablen zur Vermeidung von Namenskonflikten
 - Ersetzen der Anfrage durch Regelkörper
 - Verwenden der durch die Unifikation des Regelkopfes hergestellten Variablenbindungen
 - Abarbeiten des Regelkörpers als neue, komplexe Anfrage
- Pragmatik: Ein Regelkopf ist normalerweise keine Grundstruktur. Die Variablen des Regelkopfes können als formale Parameter einer Prozedur betrachtet werden.

erweiterte Beispieldatenbasis 1, unverändertes Prädikatsschema

```
% liebt(?Wer,?Wen-oder-was)
% Wer und Wen-oder-was sind Namen, so dass
% Wen-oder-was von Wer geliebt wird
liebt(hans,geld).
liebt(hans,susi).
liebt(susi,buch).
liebt(karl,buch).
liebt(paul,X):-liebt(X,buch).
```

Testen eines Prädikats (2)

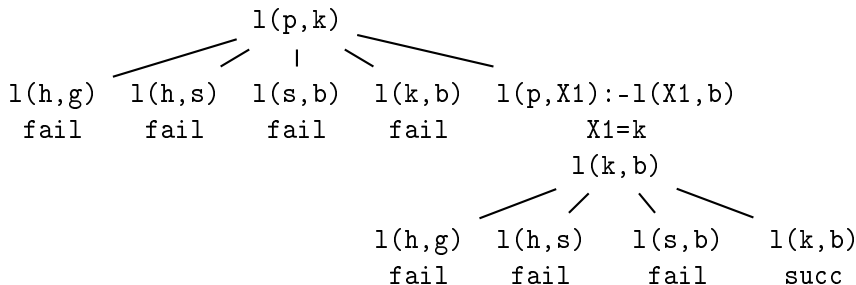
Beim systematischen Testen eines Prädikats müssen stets

- positive und negative Beispiele,
- alle Instanziierungsvarianten und
- alle wesentlichen Fallunterscheidungen berücksichtigt werden.

Beispielaufufe

- Anfrage mit vollständig instanziiertem Ziel: `liebt(+,+)`

`?- liebt(paul,karl).`



→ Test auf Enthaltensein eines Elementes in der Relation

Beispielaufufe

- Anfrage mit vollständig instanziiertem Ziel

liebt(+,+)

```
?- liebt(paul, karl).           #1
    liebt(hans, geld).         fail
    liebt(hans, susi).         fail
    liebt(susi, buch).         fail
    liebt(karl, buch).         fail
    liebt(paul, X1):-liebt(X1, buch). succ(X1=karl)
?- liebt(karl, buch).          #2
    liebt(hans, geld).         fail
    liebt(hans, susi).         fail
    liebt(susi, buch).         fail
    liebt(karl, buch).         succ

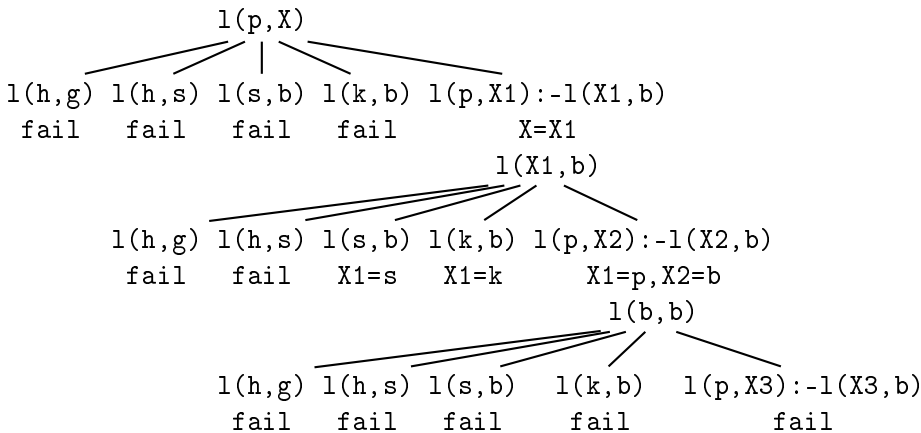
true.
```


Beispielaufufe

- Anfrage mit partiell instanziiertem Ziel (1):

liebt(+, -)

?- liebt(paul,X).



Beispielaufufe

- Anfrage mit partiell instanziiertem Ziel (1) liebt(+, -)

?- liebt(paul,X).	#1
liebt(hans,geld).	fail
...	
liebt(paul,X1):-liebt(X1,buch).	succ(X=X1)
?- liebt(X1,buch).	#2
liebt(hans,geld).	fail
liebt(hans,susi).	fail
liebt(susi,buch).	succ(X=X1=susi)
X=susi ;	
liebt(karl,buch).	succ(X=X1=karl)
X=karl ;	
liebt(paul,X2):-liebt(X2,buch).	succ(X1=paul , X2=buch)
?- liebt(buch,buch).	#3
liebt(hans,geld).	fail
...	
liebt(paul,X3):-liebt(X3,buch).	fail
BT #2	
BT #1	
false.	

Beispielaufufe

- Anfrage mit partiell instanziiertem Ziel (2)

liebt(-,+)

```
?- liebt(X,buch).                #1
    liebt(hans,geld).            fail
    liebt(hans,susi).            fail
    liebt(susi,buch).            succ(X=susi)
X=susi ;
    liebt(karl,buch).            succ(X=karl)
X=karl ;
    liebt(paul,X1):-liebt(X1,buch). succ(X=paul,X1=buch)
?- liebt(buch,buch).             #2
    liebt(hans,geld).            fail
    liebt(hans,susi).            fail
    liebt(susi,buch).            fail
    liebt(karl,buch).            fail
    liebt(paul,X2):-liebt(X2,buch). fail
BT #1
false
```

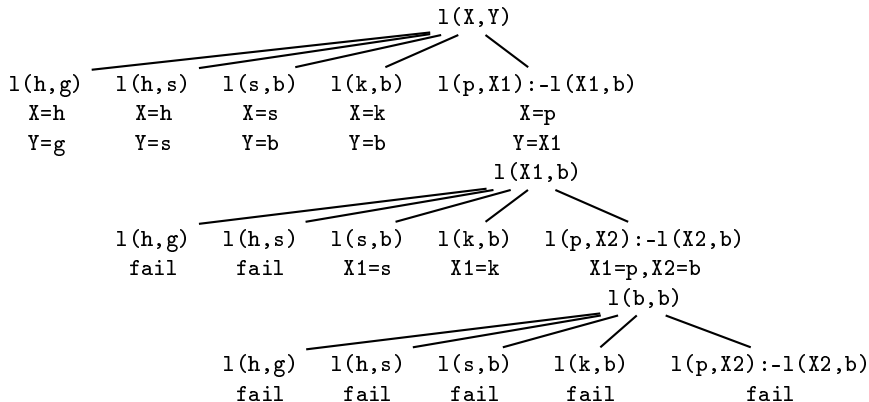
→ Aufzählung aller Elemente der Relation, die der
Anfragebedingung genügen

Beispielaufufe

- Anfrage mit vollständig uninstantiiertem Ziel

liebt(-,-)

?- liebt(X,Y).



→ vollständige Aufzählung der Relation

Beispielanfragen

- Anfrage mit vollständig uninstantiiertem Ziel

```
?- liebt(X,Y).                                #1
    liebt(hans,geld).                        succ(X=hans,Y=geld)
X=hans,Y=geld ;
...
liebt(paul,X1):-liebt(X1,buch).              succ(X=paul,Y=X1)
?- liebt(X1,buch).                          #2
    liebt(hans,geld).                      fail
    liebt(hans,susi).                     fail
    liebt(susi,buch).                     succ(X=paul,Y=X1=susi)
X=paul,Y=susi ;
    liebt(karl,buch).                      succ(X=paul,Y=X1=karl)
X=paul,Y=karl ;
    liebt(paul,X2):-liebt(X2,buch).         succ(X=paul,Y=X1=paul,X2=buch)
    ?- liebt(buch,buch).                   #3
    . . .
    BT #2
    BT #1
false.
```

Spezielle Relationstypen

- Symmetrische Relationen
- Transitive Relationen
- 1:m Relationen
- Reflexive Relationen

Symmetrische Relationen

Symmetrie

Eine zweistellige Relation p ist symmetrisch, wenn

$$p(X, Y) \leftrightarrow p(Y, X)$$

allgemeingültig ist.

Symmetrische Relationen

- Beispiel: Zwillingsrelation

```
% zwillling_von(?Pers1,?Pers2)  
% Pers1 und Pers2 sind Namen, so dass Pers1 und Pers2  
% Zwillinge sind
```

```
zwillling_von(paul,anne).  
zwillling_von(hans,nina).
```

```
?- zwillling_von(paul,anne).  
true.  
?- zwillling_von(anne,paul).  
false.
```

Eine extensional definierte, zweistellige Relation ist standardmäßig unsymmetrisch.

Symmetrische Relationen

- Herstellung der Symmetrieeigenschaft erfordert zusätzlichen Aufwand
- 3 Möglichkeiten
 1. Angabe der Inversen

```
zwillling_von(paul, anne).  
zwillling_von(anne, paul).  
zwillling_von(hans, nina).  
zwillling_von(nina, hans).
```

→ Redundanz in der Datenbasis

Symmetrische Relationen

- Herstellung der Symmetrieeigenschaft (Forts.)

2. Symmetriedefinition über ein Hilfsprädikat

```
% zwillling_von(?Pers1,?Pers2)
% Pers1 und Pers2 sind Namen, so dass Pers1
% und Pers2 Zwillinge sind
zwillling_von(X,Y):-z_v(X,Y).
zwillling_von(X,Y):-z_v(Y,X).
z_v(paul,anne).
z_v(hans,nina).
```

→ faktenbasierte Modellierung

3. intensionale Definition

```
zwillling_von(X,Y) :- mutter_von(M,X), mutter_von(M,Y),
    hat_geburtstag(X,D), hat_geburtstag(Y,D).
```

→ wissensbasierte (symmetrische) Modellierung

Symmetrische Relationen

- Hilfsprädikat: ist es wirklich erforderlich?

```
zwilling_von(paul, anne).
```

```
zwilling_von(X,Y):-zwilling_von(Y,X).
```

```
?- zwilling_von(paul, anne).  
true.
```

```
?- zwilling_von(anne, paul).  
true.
```

Symmetrische Relationen

- unterspezifizierter Aufruf: `zwillling_von(-,-)`

```
?- zwillling_von(X,Y).  
   X=paul, Y=anne ;  
   X=anne, Y=paul;  
   X=paul, Y=anne ;  
   X=anne, Y=paul .
```

- Differenz zwischen denotationeller und operationaler Semantik:
 - denotationell: Relation enthält zwei Elemente
 - operational: Suche erzeugt (zyklisch) beliebig viele Variablenbindungen

Symmetrische Relationen

- Aufruf mit einem unbekannten Objekt: `zwillling_von(+,+)`

```
?- zwillling_von(paul, karl) .
```

```
...
```

- Differenz zwischen denotationeller und operationaler Semantik:
 - denotationell: Ziel ist in der Relation nicht enthalten (Resultat: no)
 - operational: Suche terminiert nicht
- Verdacht: *rekursive* Spezifikation ruft Terminierungsprobleme hervor!
- Ausweg: Vermeiden *unbeschränkter* Rekursion durch Hilfsprädikate

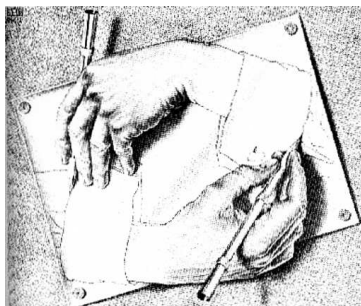
Rekursion (1)

Rekursion

Selbstbezüglichkeit einer formalen Struktur:

- zwei Formen
 - Eine Prädikatsdefinition wird auf sich selbst zurückgeführt:

```
zwillling(X,Y) :-  
    zwillling(Y,X).
```



- Ein Datentyp wird durch sich selbst definiert (Struktur, Liste).

Transitive Relationen

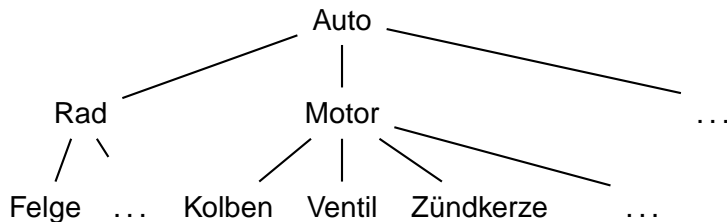
Transitivität

Eine zweistellige Relation p ist transitiv, wenn

$$p(X, Y) \wedge p(Y, Z) \rightarrow p(X, Z)$$

allgemeingültig ist.

- Verwendung z. B. zur terminologischen Wissensrepräsentation



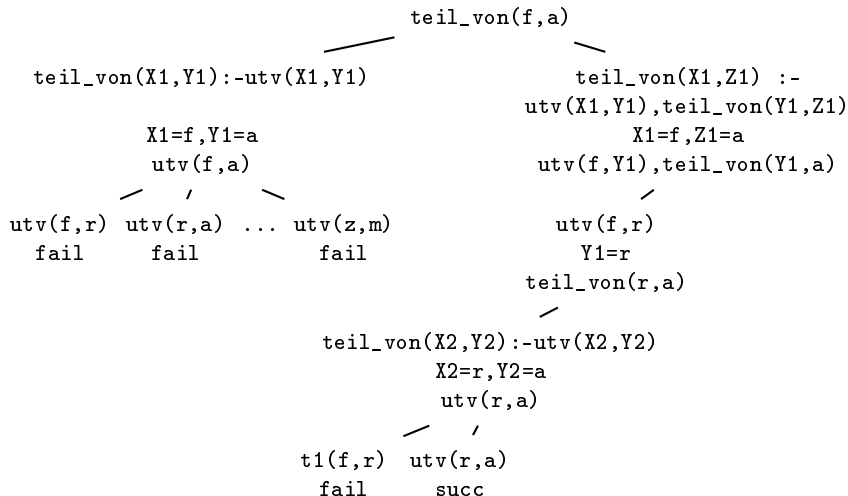
Transitive Relationen

```
% teil_von(?Teil,?Komplexes-Teil)
% Teil und Komplexes-Teil sind Namen, so dass Teil
% Bestandteil von Komplexes-Teil ist
teil_von(X,Y):-utv(X,Y).
teil_von(X,Z):-utv(X,Y),teil_von(Y,Z).
utv(folge,rad).
utv(rad,auto).
utv(motor,auto).
utv(kolben,motor).
utv(ventil,motor).
utv(zuendkerze,motor).
```


Transitive Relationen

- vollständig spezifizierte Anfrage: Relationstest `teil_von(+,+)`

?- `teil_von(folge,auto).`



Transitive Relationen

- sonstige Anfragen:

```
?- teil_von(bremse,auto).  
false.
```

```
?- teil_von(felge,X).  
X=rad ;  
X=auto.
```

```
?- teil_von(X,rad).  
X=felge.
```

```
?- teil_von(X,Y).  
X=felge, Y=rad ;  
...  
X=zuendkerze, Y=auto.
```

Funktionale Abhängigkeit

Eine zweistellige Relation $p(X, Y)$ heißt

- vom Typ 1:m, wenn $p(X, Y) \wedge p(Z, Y) \rightarrow X = Z$ allgemeingültig ist.
- vom Typ m:1, wenn $p(X, Y) \wedge p(X, Z) \rightarrow Y = Z$ allgemeingültig ist.

- “funktionale” Spezifikation einer Argumentposition:
Ist $p(X, Y)$ eine Relation vom Typ 1:m, dann gilt $X = f(Y)$.
- 1:m- und m:1-Relationen sind Spezialfälle einer m:n-Relation

1:m-Relationen

- Beispiele für 1:m-Relationen:
 - `mutter_von(Mutter, Tochter)`
 - `alter_von(Jahre, Person)`
- Beispiele für m:1-Relationen:
 - `utv(Obj1, Obj2)`
 - `liegt_in(Stadt, Land)`

1:m-Relationen

Prädikatsdefinitionen sind im allgemeinen Fall vom Typ m:n.

- auch bei eindeutigen Relationen wird auf Anfrage immer nach alternativen Funktionswerten gesucht
 - unnötiger Suchaufwand
 - u.U. Terminierungsprobleme bei rekursiven Definitionen
 - möglicher Ausweg:
Abbruch der Suche nach dem ersten Resultat $\rightarrow !/0$ (cut)

Reflexive Relationen

Reflexivität

Eine zweistellige Relation p ist reflexiv, wenn

$$p(X, X)$$

allgemeingültig ist.

- Beispiel: Gleichheit, Unifizierbarkeit

$$X = Y \Leftrightarrow Y = X$$

Intensionale Relationsdefinitionen, die sich (indirekt) auf eine Gleichheitsrelation (z.B. Unifizierbarkeit) beziehen, sind standardmäßig reflexiv.

Reflexive Relationen

```
% schwester_von(?Schwester,?Person)
schwester_von(X,Y):-kind_von(X,Z),kind_von(Y,Z),weiblich(X).

% kind_von(?Kind,?Elternteil)
kind_von(susi,anne).
kind_von(jane,anne).
kind_von(karl,anne).

weiblich(susi).
weiblich(jane).
weiblich(anne).
maennlich(karl).

?- schwester_von(S,susi).
S=susi ;
S=jane.
```

Anwendungsbeispiel: Wegplanung

- Basis: Erreichbarkeitsrelation

```
% con(?Ort1,?Ort2)
```

```
% Ort1 und Ort2 sind Namen, so dass Ort2
```

```
% von Ort1 aus erreichbar ist
```

```
con(hamburg,berlin).
```

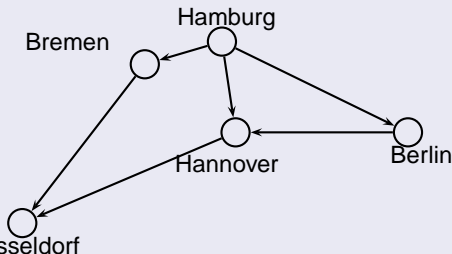
```
con(hamburg,hannover).
```

```
con(hamburg,bremen).
```

```
con(bremen,duesseldorf).
```

```
con(berlin,hannover)
```

```
con(hannover,duesseldorf)
```



Anwendungsbeispiel: Wegplanung

- Symmetrieeigenschaft: Eine extensional definierte, zweistellige Relation ist standardmäßig unsymmetrisch.

```
?- con(hamburg,bremen) .
```

```
    true.
```

```
?- con(bremen,hamburg) .
```

```
    false.
```

Anwendungsbeispiel: Wegplanung

- Symmetrie durch intensionale Spezifikation mit Hilfsprädikat

```
% con_sym(?Ort1,?Ort2)
% Ort1 und Ort2 sind Namen, so dass Ort1
% und Ort2 wechselseitig erreichbar sind
con_sym(X,Y):-con(X,Y).
con_sym(X,Y):-con(Y,X).
con(hamburg,berlin).
con(hamburg,hannover).
...
```

```
?- con_sym(hamburg,hannover).
   true.
?- con_sym(hannover,hamburg).
   true.
```

Anwendungsbeispiel: Wegplanung

- Transitivität durch intensionale Spezifikation mit Hilfsprädikat

```
% con_trans(?Ort1,?Ort2)
% Ort1 und Ort2 sind Namen, so dass zwischen Ort1 und Ort2
% eine (unsymmetrische) transitive Erreichbarkeit besteht
con_trans(X,Y):-con(X,Y).
con_trans(X,Y):-con(X,Z),con_trans(Z,Y).
con(hamburg,berlin).
con(hamburg,hannover).
...
```

Anwendungsbeispiel: Wegplanung

```
?- con_trans(hamburg,duesseldorf).  
    true ;  
    true ;  
    true ;  
    false.  
  
?- con_trans(hamburg,X).  
    X=berlin ;  
    X=hannover ;  
    X=bremen ;  
    X=hannover ;  
    X=duesseldorf ;  
    X=duesseldorf ;  
    X=duesseldorf ;  
    false.
```

Anwendungsbeispiel: Wegplanung

- Kopplung von Transitivität und Symmetrie ???

```
% con_sym_trans(Ort1,Ort2)
% Ort1 und Ort2 sind Namen, so dass zwischen Ort1 und Ort2
% eine symmetrische und transitive Erreichbarkeit besteht
con_sym_trans(X,Y):-con(X,Y).
con_sym_trans(X,Y):-con(Y,X).
con_sym_trans(X,Y):-con(X,Z),con_sym_trans(Z,Y).
con_sym_trans(X,Y):-con(Z,X),con_sym_trans(Z,Y).
...
```

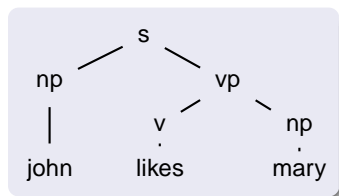
```
?- con_sym_trans(hannover,bremen).
    true ;
    true ;
    true ;
    ...
?- con_sym_trans(duesseldorf,berlin).
    ...
```

Fazit: Deduktive Datenbanken

- Berechnung einer Relation statt Aufzählung
- Vermeiden von Redundanz
- Spezifikation von Relationen über unendlichen Domänen
- transitive Hülle einer Relation ist berechenbar
- aber: Terminierungsprobleme bei rekursiven Definitionen
- Effizienzprobleme bei persistenter Datenhaltung
- eingeschränktes Modell ist Bestandteil von SQL-99

→ GDB

Rekursive Datenstrukturen



5. Rekursive Datenstrukturen

Eingebettete Strukturen

Arithmetik, relational

Operatorstrukturen

Arithmetik, funktional

Überblick Prolog-Syntax

Datenbasis	::=	Klausel {Klausel}
Prozedur	::=	Klausel {Klausel}
Ziel	::=	elementares_Ziel komplexes_Ziel
elementares_Ziel	::=	Struktur
komplexes_Ziel	::=	elementares_Ziel (';' ',') Ziel
Klausel	::=	(Fakt Ziel Regel) ''
Fakt	::=	Struktur
Regel	::=	Kopf ':-' Körper
Kopf	::=	Struktur
Körper	::=	Ziel
Struktur	::=	Name['('Term{','Term}')'] Operatorausdruck
Term	::=	Konstante Variable Struktur Liste
Konstante	::=	Zahl Name gequoteter_Name
Name	::=	Kleinbuchstabe {alphanumerisches_Symbol}
Variable	::=	benannte_Variable unbenannte_Variable
benannte_Variable	::=	Großbuchstabe {alphanumerisches_Symbol}
unbenannte_Variable	::=	'_'

Überblick Prolog-Syntax

Datenbasis	::=	Klausel {Klausel}
Prozedur	::=	Klausel {Klausel}
Ziel	::=	elementares_Ziel komplexes_Ziel
elementares_Ziel	::=	Struktur
komplexes_Ziel	::=	elementares_Ziel (';' ',') Ziel
Klausel	::=	(Fakt Ziel Regel) ''
Fakt	::=	Struktur
Regel	::=	Kopf ':-' Körper
Kopf	::=	Struktur
Körper	::=	Ziel
Struktur	::=	Name['('Term{'Term'}')'] Operatorausdruck
Term	::=	Konstante Variable Struktur Liste
Konstante	::=	Zahl Name gequoteter_Name
Name	::=	Kleinbuchstabe {alphanumerisches_Symbol}
Variable	::=	benannte_Variable unbenannte_Variable
benannte_Variable	::=	Großbuchstabe {alphanumerisches_Symbol}
unbenannte_Variable	::=	'_'

Überblick Prolog-Syntax

Datenbasis	::=	Klausel {Klausel}
Prozedur	::=	Klausel {Klausel}
Ziel	::=	elementares_Ziel komplexes_Ziel
elementares_Ziel	::=	Struktur
komplexes_Ziel	::=	elementares_Ziel (';' ',') Ziel
Klausel	::=	(Fakt Ziel Regel) ''
Fakt	::=	Struktur
Regel	::=	Kopf ':-' Körper
Kopf	::=	Struktur
Körper	::=	Ziel
Struktur	::=	Name['('Term{'','Term'})'] Operatorausdruck
Term	::=	Konstante Variable Struktur Liste
Konstante	::=	Zahl Name gequoteter_Name
Name	::=	Kleinbuchstabe {alphanumerisches_Symbol}
Variable	::=	benannte_Variable unbenannte_Variable
benannte_Variable	::=	Großbuchstabe {alphanumerisches_Symbol}
unbenannte_Variable	::=	'_'

Überblick Prolog-Syntax

Datenbasis	::=	Klausel {Klausel}
Prozedur	::=	Klausel {Klausel}
Ziel	::=	elementares_Ziel komplexes_Ziel
elementares_Ziel	::=	Struktur
komplexes_Ziel	::=	elementares_Ziel (';' ',') Ziel
Klausel	::=	(Fakt Ziel Regel) ''
Fakt	::=	Struktur
Regel	::=	Kopf ':-' Körper
Kopf	::=	Struktur
Körper	::=	Ziel
Struktur	::=	Name['('Term{'','Term'}')'] Operatorausdruck
Term	::=	Konstante Variable Struktur Liste
Konstante	::=	Zahl Name gequoteter_Name
Name	::=	Kleinbuchstabe {alphanumerisches_Symbol}
Variable	::=	benannte_Variable unbenannte_Variable
benannte_Variable	::=	Großbuchstabe {alphanumerisches_Symbol}
unbenannte_Variable	::=	'_'

Eingebettete Strukturen

- Strukturen können rekursiv eingebettet sein
- Prolog-Datenbanken müssen nicht in 1. Normalform sein (NF²)

```
angebot(produkt('tm-416'),  
        kategorie(radio),  
        status(am_lager),  
        preis(euro(100))).
```

- wichtiger Spezialfall: rekursive Selbsteinbettung einer Struktur

```
s(s(s(0)))
```

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

Eingebettete Strukturen

- z.B. Verwendung zum typischeren Programmieren

```
angebot(produkt(radio),euro(100)).
```

```
?- anbot(Produkt,euro(Preis)), Preis < 200.  
Produkt=produkt(radio),Preis=100.
```

- Datenabstraktion: Zusammenfassung von elementaren zu komplexen Strukturen

```
adresse(plz(22527),  
        strasse('Vogt-Kölln-Straße'),  
        ort('Hamburg'))
```

Eingebettete Strukturen

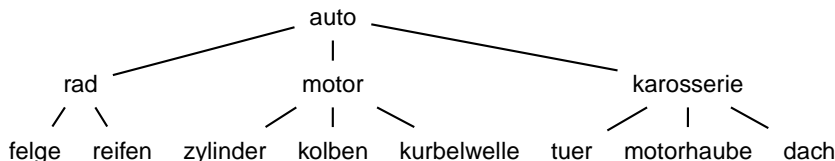
- rekursiv eingebettete Strukturen entsprechen einer Baumstruktur

Strukturen	\Leftrightarrow	Verzweigungen des Baumes (Knoten)
Prädikatsnamen	\Leftrightarrow	Knotenmarkierungen
äußerster Prädikatsname	\Leftrightarrow	Wurzelknoten des Baumes
Argumente	\Leftrightarrow	Kanten
Atome als Argument	\Leftrightarrow	Blattknoten

Baumstrukturen

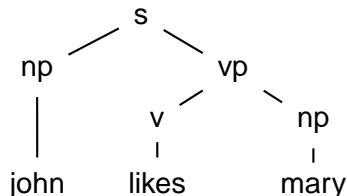
- Teil-von-Hierarchien

```
auto(rad(felge,reifen),  
      motor(zyylinder,kolben,kurbelwelle),  
      karosserie(tuer,motorhaube,dach)).
```



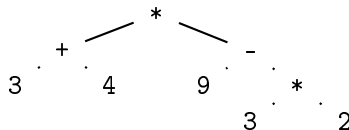
- Syntaxbäume in der natürlichen Sprache

```
s(np(john),  
  vp(v(likes),  
      np(mary)))
```



Baumstrukturen

- Syntaxbäume für Programmiersprachen
 - Arithmetischer Ausdruck: $(3 + 4) * (9 - 3 * 2)$
 - in Präfixnotation:
 $*(+(3,4),$
 $-(9,$
 $*(3,2)))$



Unifikation von komplexen Strukturen

Ermitteln einer solchen Variablensubstitution, die die Gleichheit von *zwei Strukturen* herstellt.

- Vergleichskriterien:
 - identischer Prädikatsname
 - gleiche Stelligkeit
 - *rekursive* Unifikation der Argumente

Struktur / Struktur	Unifikation	$a(X, c) = a(b, Y)$
Variable / Struktur	Instanziierung	$X = a(b, c)$
Struktur / Variable	Instanziierung	$d(e, f) = Y$
Variable / Variable	Koreferenz	$X = Y$

Unifikation III

- Beispiele für die Unifikation von Strukturen

Struktur ₁	Struktur ₂	Variablensubstitution
$a(X, c)$	$a(b, Y)$	$X=b, Y=c$
$a(X, X)$	$a(d(Y), d(b))$	$X=d(b), Y=b$
$a(X, a(X))$	$a(Y, Y)$	$X=Y=a(X)$

- Problemfall: Koreferenz über mehrere Rekursionsebenen hinweg
→ Aufbau unendlicher durch Unifikation endlicher Strukturen.

?- $a(X)=X$.

$X = a(X)$.

?- $a(X)=X, \text{write}(X)$.

$a(**)$

$X = a(X)$

?- $a(X)=X, X=a(A), X=a(a(B)), X= a(a(a(C)))$.

$X = A, A = B, B = C, C = a(C)$.

Unifikation III

- Pragmatik: Unifikation ist gleichzeitig
 - a) Testoperator:
Sind die zu unifizierenden Strukturen miteinander verträglich?
 - b) Accessor (Selektor, Observer):
Aus welchen Bestandteilen setzt sich eine komplexe Struktur zusammen?
 - c) Konstruktor:
Setze eine komplexe Struktur aus gegebenen Bausteinen zusammen!

Arithmetik, relational

- Axiomatisierung der Arithmetik für natürliche Zahlen
- Giuseppe PEANO, italienischer Mathematiker, 1858-1932

natürliche Zahl

Anfangselement:	0 ist eine natürliche Zahl.
Nachfolgerbeziehung:	Jede natürliche Zahl n besitzt einen unmittelbaren Nachfolger $s(n)$, der ebenfalls eine natürliche Zahl ist.
Domänenabschluss:	Nur die so gebildeten Objekte sind natürliche Zahlen.

→ rekursive Definition!

Arithmetik, relational

- Daten: PEANO-Terme
- Syntax: $\text{PEANO-Term} ::= '0' \mid 's' (\text{PEANO-Term})$
- Semantik:

Term	arithmetische Interpretation
0	0
s(0)	1
s(s(0))	2
s(s(s(0)))	3
...	...

- PEANO-Terme sind kein Prolog-Konstrukt, sondern eine spezielle Form von Prolog-Strukturen

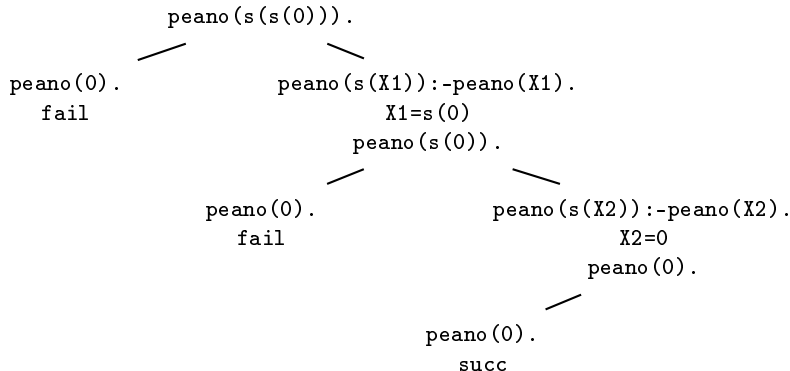
- Programme: Prädikate über PEANO-Termen
- z.B. Typtest für PEANO-Zahlen: `peano/1`

```
% peano(+Term)
% Term ist ein Peano-Term
peano(0).                               % Rekursionsabschluss
peano(s(X)) :- peano(X).                % Rekursionsschritt
```

→ Rekursive Definition

Beispielanfragen

- vollständig instantiiertes Aufruf: `peano(+)`



Beispielanfragen

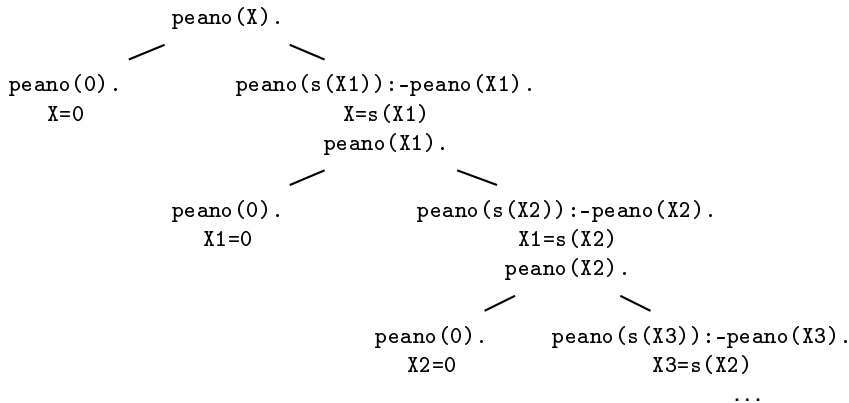
- vollständig instanziiertes Aufruf

?- peano(s(s(0))).	#1
peano(0).	fail
peano(s(X1)):-peano(X1).	succ(X1=s(0))
?- peano(s(0)).	#2
peano(0).	fail
peano(s(X2)):-peano(X2).	succ(X2=0)
?- peano(0).	#3
peano(0).	succ
true.	
?- peano(susi).	#1
peano(0).	fail
peano(s(X1)):-peano(X1).	fail
false.	

- Konsumtion einer rekursiven Struktur auf der Argumentposition beim rekursiven Abstieg

Beispielanfragen

- unterspezifizierter Aufruf: `peano(-)`



- Prädikatsschema kann verallgemeinert werden: `peano(?Term)`

Beispielanfragen

- Unterspezifizierter Aufruf

?- peano(X).	#1
peano(0).	succ(X=0)
X=0 ;	
peano(s(X1)):-peano(X1).	succ(X=s(X1))
?- peano(X1).	#2
peano(0).	succ(X1=0)
X=s(0) ;	
peano(s(X2)):-peano(X2).	succ(X1=s(X2))
?- peano(X2).	#3
peano(0).	succ(X2=0)
X=s(s(0)) ;	
...	

- Konstruktion einer rekursiven Struktur auf der Argumentposition beim rekursiven Abstieg

Rekursion (2)



BAUER/GOOS, 1982

- Rekursion (2): selbstbezügliche Definition eines Prädikats
 - unter *Konsumtion einer Datenstruktur* auf einer Argumentposition und
 - mit einer Terminierungsbedingung (Rekursionsabschluss) für diese Struktur
- wichtiger Spezialfall: Endrekursion

Endrekursion

- Berechnung erfolgt nur beim rekursiven Abstieg.
 - Endergebnis ist am Rekursionsabschluss bereits vollständig bekannt
 - Zwischenergebnisse müssen nicht mehr auf dem Stack der abstrakten Maschine aufbewahrt werden
 - Compilation in effizienten (weil iterativen) Code möglich
-
- Endrekursion liegt vor, wenn der rekursive Aufruf das letzte Teilziel im Rekursionsschritt ist.

Arithmetik, relational

- Vergleich von PEANO-Zahlen: `lt(Peano_Zahl1, Peano_Zahl2)`
(less than)
- gewünschtes Verhalten:

	Ziel	Resultat		Ziel	Resultat
1	<code>lt(0,s(0))</code>	<code>true.</code>	4	<code>lt(0,0)</code>	<code>false.</code>
2	<code>lt(0,s(s(0)))</code>	<code>true.</code>	5	<code>lt(s(0),0)</code>	<code>false.</code>
3	<code>lt(s(0),s(s(0)))</code>	<code>true.</code>	6	<code>lt(s(s(0)),s(0))</code>	<code>false.</code>

- rekursive Definition

```
% lt(?Term1,?Term2)
% Term1 und Term2 sind Peano-Terme, so dass Term1
% kleiner als Term2
lt(0,s(_)).
lt(s(X),s(Y)):-lt(X,Y).
```

Beispielanfragen

- vollständig spezifizierte Aufrufe: Konsumtion von rekursiven Datenstrukturen auf beiden Argumentpositionen
- unterspezifizierte Aufrufe

```
?- lt(X,s(s(0))).           #1
    lt(0,s(_)).             succ(X=0)
X=0 ;
    lt(s(X1),s(Y1)):-lt(X1,Y1).    succ(X=s(X1),Y1=s(0))
?- lt(X1,s(0)).              #2
    lt(0,s(_)).              succ(X1=0)
X=s(0) ;
    lt(s(X2),s(Y2)):-lt(X2,Y2).    succ(X1=s(X2),Y2=0)
?- lt(X2,0).                 #3
    lt(0,s(_)).              fail
    lt(s(X3),s(Y3)):-lt(X3,Y3).    fail
    BT #2
    BT #1
false.
```

Beispielanfragen

- unterspezifizierte Aufrufe:

?- lt(s(s(0)),X).	#1
lt(0,s(_)).	fail
lt(s(X1),s(Y1)):-lt(X1,Y1).	succ(X1=s(0),X=s(Y1))
?- lt(s(0),Y1).	#2
lt(0,s(_)).	fail
lt(s(X2),s(Y2)):-lt(X2,Y2).	succ(X2=0,Y1=s(Y2))
?- lt(0,Y2).	#3
lt(0,s(_)).	succ(Y2=s(_))
X=s(s(s(_))) ;	
lt(s(X3),s(Y3)):-lt(X3,Y3).	fail
BT #2	
BT #1	
false.	

unterspezifiziertes Resultat

generische Beschreibung einer unendlich großen Resultatsmenge
Das Ziel wird aufgrund der vorhandenen Information soweit wie möglich instanziiert.

Indirekte Rekursion

- Test auf gerade/ungerade PEANO-Zahlen: `even(Peano_Zahl)`,
`odd(Peano_Zahl)`
- gewünschtes Verhalten:

	Ziel	Resultat		Ziel	Resultat
1	<code>even(0)</code>	<code>true.</code>	4	<code>odd(0)</code>	<code>false.</code>
2	<code>even(s(0))</code>	<code>false.</code>	5	<code>odd(s(0))</code>	<code>true.</code>
3	<code>even(s(s(0)))</code>	<code>true.</code>	6	<code>odd(s(s(0)))</code>	<code>false.</code>
4	<code>even(s(s(s(0))))</code>	<code>false.</code>	6	<code>odd(s(s(s(0))))</code>	<code>true.</code>

- rekursive Definitionen

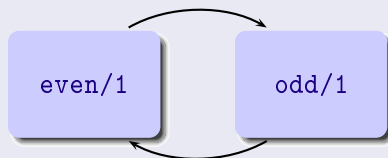
```
% even(?Term), odd(?Term)
% Term ist ein geradzahliger/ungeradzahliger Peano-Term
even(0).
even(s(P)) :- odd(P).

odd(s(P)) :- even(P).
```

Indirekte Rekursion

indirekte Rekursion

Der rekursive Aufruf erfolgt nicht unmittelbar in der Klausel, die das Prädikat definiert, sondern in einem zweiten Prädikat, das von Ersterem aufgerufen wird.



Indirekte Rekursion kann auch über mehr als eine Zwischenstufe erfolgen

Arithmetik, relational

- Addition von PEANO-Zahlen: $\text{add}(\text{Summand}_1, \text{Summand}_2, \text{Summe})$
- gewünschtes Verhalten:

1	$\text{add}(0, 0, 0)$	true.	5	$\text{add}(0, s(0), 0)$	false
2	$\text{add}(0, s(0), s(0))$	true.	6	$\text{add}(s(0), s(0), s(0))$	false
3	$\text{add}(s(0), 0, s(0))$	true.			
4	$\text{add}(s(s(0)), s(0), s(s(s(0))))$	true.			

- rekursive Definition:

```
% add(?Summand1, ?Summand2, ?Summe)
% Summand1, Summand2 und Summe sind Peano-Terme,
% so dass gilt: Summand1 + Summand2 = Summe
add(0, X, X).                               % Rekursionsabschluss
add(s(X), Y, s(R)) :- add(X, Y, R).         % Rekursionsschritt
```

Beispielanfragen

- **unterspezifizierte Aufrufe:** `add(+, +, -)` z.B. `2+1=Summe`

```
?- add(s(s(0)),s(0),Sum) .           #1
   add(0,X1,X1) .                   fail
   add(s(X1),Y1,s(R1)):-add(X1,Y1,R1) . succ(X1=s(0),Y1=s(0),Sum=s(R1))
?- add(s(0),s(0),R1) .              #2
   add(0,X2,X2) .                   fail
   add(s(X2),Y2,s(R2)):-add(X2,Y2,R2) . succ(X2=0,Y2=s(0),R1=s(R2))
?- add(0,s(0),R2) .                 #3
   add(0,X3,X3) .                   succ(X3=s(0)=R2)

Sum=s(s(s(0))) ;
   add(s(X3),Y3,s(R3)):-add(X3,Y3,R3) .   fail
BT #2
BT #1
false.
```

Beispielanfragen

- unterspezifizierte Aufrufe: `add(+, -, +)`

z.B. `2+Smd=3`

<pre>?- add(s(s(0)),Smd,s(s(s(0)))) . add(0,X1,X1) . add(s(X1),Y1,s(R1)) :- add(X1,Y1,R1) . ?- add(s(0),Y1,s(s(0))) . add(0,X2,X2) . add(s(X2),Y2,s(R2)) :- add(X2,Y2,R2) . ?- add(0,Y2,s(0)) . add(0,X3,X3) . Smd=s(0) ; add(s(X3),Y3,s(R3)) :- add(X3,Y3,R3) . BT #2 BT #1 false.</pre>	<pre>#1 fail succ(X1=s(0),Smd=Y1, R1=s(s(0))) #2 fail succ(X2=0,Y1=Y2,R2=s(0)) #3 succ(Y2=X3=s(0)) fail</pre>
---	--

Beispielanfragen

- weitere unterspezifizierte Aufrufe:

```
?- add(Smd,s(0),s(s(s(0)))) .      % Smd + 1 = 3  
Smd=s(s(0)) .
```

```
?- add(Smd1,Smd2,s(s(0))) .        % Smd1 + Smd2 = 3  
Smd1=0, Smd2=s(s(0)) ;  
Smd1=s(0), Smd2=s(0) ;  
Smd1=s(s(0)), Smd2=0 .
```

Testen eines Prädikats (3): Regressionstests

1. Speichern aller Testfälle als Fakten
2. Ableiten der Prädikatsdefinition aus den Testbeispielen
3. teilautomatisierte Überprüfung mit einfacher Testroutine
4. Testen der unterspezifizierten Aufrufe
5. sukzessives Erweitern der Testdatenbank

- positive und negative Testfälle, z.B.

<code>test_daten(add(0,0,0)).</code>	<code>% positiv</code>
<code>test_daten(add(0,s(0),s(0))).</code>	<code>% positiv</code>
<code>test_daten(add(s(0),0,s(0))).</code>	<code>% positiv</code>
<code>test_daten(add(s(s(0)),s(0),s(s(s(0))))).</code>	<code>% positiv</code>
<code>test_daten(add(0,s(0),0)).</code>	<code>% negativ</code>
<code>test_daten(add(s(0),s(0),s(0))).</code>	<code>% negativ</code>

Exkurs: Programmiermethodik

- teilautomatisierte Überprüfung mit einfacher Testroutine, z.B.

```
% Testroutine
% ruft Testbeispiele der Form test_daten(call) auf

test :- test_daten(X), write(X), diagnose(X), fail.

diagnose(X) :- X, write_ln(' yes').
diagnose(X) :- not(X), write_ln(' no').

?- test.
add(0, 0, 0) yes
add(0, s(0), s(0)) yes
add(s(0), 0, s(0)) yes
add(s(s(0)), s(0), s(s(s(0)))) yes
add(0, s(0), 0) no
add(s(0), s(0), s(0)) no
false.
```


Operatoren

- erlauben es dem *Nutzer*, die Syntax der Sprache zu modifizieren:
 - übersichtliche Schreibweise
 - Nähe zu etablierten Notationen (z.B. Mathematik)
- sind syntaktische Varianten für ein- oder zweistellige Strukturen:
$$A \text{ op } B \Leftrightarrow \text{op}(A, B)$$
- haben keine Semantik und definieren keine eigenständigen Prädikate!

Operatoren

- Syntax: Operatorausdruck ::= Präfixoperator | Infixoperator | Postfixoperator
 Präfixoperator ::= Operator Operand
 Infixoperator ::= Operand Operator Operand
 Postfixoperator ::= Operand Operator
- Operatortyp: Präfixoperator Infixoperator Postfixoperator
 fx yfx xf
 lg 10 3 + 4 4!

Operatoren

- Operatorpräzedenz

→ SE I

- numerische Angabe zur Disambiguierung von Operatorausdrücken mit mehr als einem Operator

$$a + b * c \equiv a + (b * c)$$

- Wertebereich ist implementationsabhängig (1 ... 1200)
- geringere Werte binden stärker:

+	-	*	/
500	500	400	400

Operatoren

- symmetrische Operatoren (nicht-assoziativ): `xfx xf fx`
- keine Operatoreinbettung möglich
- `1 < 2 < 3` ist syntaktisch unzulässig
- Beispiel: `liebt/2`
- Verwendung als Klauselkopf

```
: -op(300,xfx,liebt).  
hans liebt geld.  
liebt(hans,susi).
```

```
?- listing.  
    hans liebt geld.  
    hans liebt susi.  
    true.
```

Operatoren

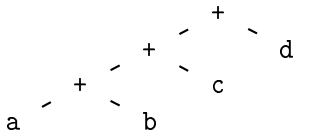
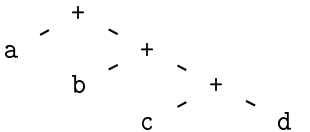
- Verwendung als Klauselkörper

```
?- X liebt susi.  
   X=hans.
```

```
?- liebt(X,geld).  
   X=hans.
```

Spezielle Operatoren

- asymmetrische Operatoren erlauben rekursive Einbettung
- Beispiel: $a + b + c + d$

links-assoziativ	rechts-assoziativ
$yfx \quad yf$	$xfy \quad fy$
$'+' ('+' ('+' (a,b) , c) , d)$	$'+' (a, '+' (b, '+' (c,d))))$
	

Spezielle Operatoren

	Präzedenz	Typ	Operator
Klauselstruktur	1200	xfx	:-
	1100	xfy	;
	1000	xfy	,
Direktiven	1200	fx	:- ?-
Negation	900	fy	\+, not
Unifikation	700	xfx	= \=
strukturelle Identität	700	xfx	== \==
strukturelle Gleichheit	700	xfx	=@= \=@=
numerischer Vergleich	700	xfx	> < >= <= =\= =:=
struktureller Vergleich	700	xfx	@> @< @>= @=<
arithmetische Funktionen	500	yfx	+ -
	400	yfx	* / //
	300	xfx	mod
	200	xfy	^

Spezielle Operatoren

- Klauselsyntax ist Operatorausdruck

```
grossvater_von(G,E):-  
    vater_von(G,V),  
    vater_von(V,E).
```

ist vollständig äquivalent zu

```
' :- ' (grossvater_von(G,E),  
        ', ' (vater_von(G,V),vater_von(V,E))) .
```

- Klammerstruktur von komplexen Zielen (Konjunktion und Disjunktion) bzw. komplexen arithmetischen Funktionen wird durch mehrstufige Präferenzen festgelegt

Spezielle Operatoren

- struktureller Vergleich: Standardordnung
 - Variable @< Atome (@< Zahl) @< Struktur
 - Atome: lexikalische Sortierung nach Zeichencode
 - Zahlen: Sortierung nach numerischem Wert (ohne Typunterscheidung)
 - Strukturen: Sortierung nach Funktor, Stelligkeit und Argumenten
- Beispiele

A @< a	true	a(b) @< b(c)	true
aaa @< abc	true	a(b) @< a(b,c)	true
a @< a(b)	true	a(b,c) @< a(b,d)	true

- wichtiger Spezialfall: lexikalischer Vergleich

Operatoren

- Strukturen als einzig verfügbare Basisrepräsentation der Logikprogrammierung
- Operatoren sind nur syntaktische Notationsvarianten.
- Programme und Daten sind syntaktisch nicht unterscheidbar
- Prädikatsdefinitionen und Prädikatsaufrufe können dynamisch erzeugt werden
- wechselseitige Umwandlung zwischen Daten und Programmen ist möglich
 - Aufruf von Daten als Programm: `call(+Struktur)`
 - Interpretation von Programmklauseln als Daten:
`clause(?Head,?Body)`

Arithmetik, funktional

- Operatorausdrücke und Strukturen stehen nur für sich selbst und haben im relationalen Verarbeitungsmodell keinen Wert
- arithmetische Relationen beschreiben Beziehung zwischen Operanden *und* gewünschtem Berechnungsergebnis

arithmetische Operatorstruktur	arithmetisches Prädikat
$+(2, 3)$	<code>add(2, 3, R) .</code>
$*(4, 6)$	<code>mult(4, 6, R) .</code>

- Arithmetische Ausdrücke müssen erst in eine Auswertungsumgebung gestellt werden
→ lokales funktionales Verarbeitungsmodell im relationalen Paradigma

arithmetische Auswertungsumgebung

spezieller Infixoperator, der den (arithmetischen) Wert von funktionalen Ausdrücken ermittelt.

- Syntax:
Auswertungsumgebung ::= Zahl 'is' Arithmetischer_Ausdruck
- Semantik: Der Wert des Ausdrucks auf der rechten Seite wird mit dem Ausdruck auf der linken Seite unifiziert.

Arithmetische Auswertungsumgebung

- Auswertung greift auf die numerischen Operationen der Basismaschine zurück
- Konsequenzen
 - linke Seite sollte keine Struktur, nur Variable oder numerische Konstante sein
 - die rechte Seite muss sich zu einem arithmetischen Wert auswerten lassen
 - die rechte Seite darf Variable nur dann enthalten, wenn diese bereits instantiiert sind
 - arithmetische Ausdrücke und Ordnungsprädikate sind nicht mehr richtungsunabhängig

Arithmetische Auswertungsumgebung

- `is/2` schlägt fehl, wenn
 - eine der Typrestriktionen für die Argumente verletzt ist
 - z.B. linke Seite ist ein komplexer Term
 - linke Seite und Wert der rechten Seite nicht miteinander unifizieren
 - z.B. Ungleichheit der numerischen Werte auf der rechten und der linken Seite
- `is/2` bricht mit Fehler ab, wenn
 - sich die rechte Seite nicht auswerten lässt, weil sie
 - kein arithmetischer Ausdruck (function) ist
 - uninstantiierte Variable enthält
- `is/2` ist mehr als die Ergibtanweisung imperativer Sprachen (`:=`)

Arithmetische Auswertungsumgebung

- Beispiele

3 is 2 * 5 - 7	succ
X is 2 * 3 + 1	succ(X=7)
2 + 1 is 2 * 5 - 7	fail
X is 5, X is 2 * 6	fail
X is 1, X is X + 1	fail
X is 1, X is X * 1	succ(X=1)
X is Y + 1	ERROR
X is a	ERROR

Weitere Auswertungsumgebungen

- arithmetische Vergleichsoperatoren: `==` `<` `>`
- funktionale Auswertung auf beiden Argumentpositionen

```
?- 10 / 2 == 4 + 1.
```

```
    true.
```

```
?- 10 - 2 == 4 * 1.
```

```
    false.
```

- können nicht zur Variableninstanziierung verwendet werden

```
?- X == 2 * 3 - 1.
```

```
    ERROR: Arguments are not sufficiently instantiated
```


- hybride Sprache
 - Funktionale Auswertung in einer relationalen Umgebung
 - Restriktionen spiegeln die nichtdeklarative Semantik der imperativen Hardwarearchitektur wieder
- Ursache: Behandlung der Arithmetik mit außerlogischen Mitteln

- Induktive Definition (funktional)

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ (n-1)! * n & \text{sonst} \end{cases}$$

Induktionsanfang
Induktionsschritt

- Übertragung in eine relationale Definition

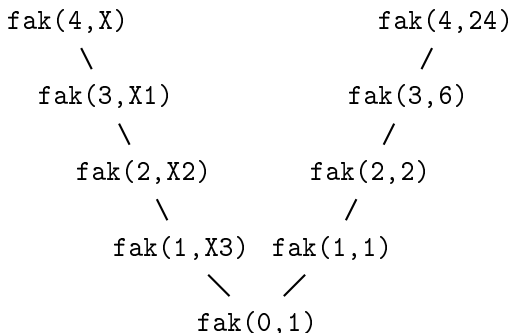
$$\text{fak}(n, r) \leftarrow \begin{cases} r = 1 & \text{für } n = 0 \\ \text{fak}(n-1, r_1), r = r_1 * n & \text{sonst} \end{cases}$$

- Übertragung in ein relationales Programm

$$\text{fak}(n, r) \leftarrow \begin{cases} r = 1 & \text{für } n = 0 \\ \text{fak}(n-1, r_1), r = r_1 * n & \text{sonst} \end{cases}$$

```
% fak(+NatZahl,?Resultat)
% NatZahl und Resultat sind natuerliche Zahlen, so dass
% Resultat = fakultaet(NatZahl)
fak(0,1).                % Rekursionsabschluss
fak(N,FakN) :-           % Rekursionsschritt
    N > 0,                % nur fuer natuerliche Zahlen
    N1 is N - 1,          % Konsumtion des Eingabearguments
    fak(N1,FakN1),        % rekursiver Aufruf
    FakN is FakN1 * N.    % Aufbau des Resultats
```

- Rekursionsschema



- eigentliche Berechnung erfolgt nur beim rekursiven Aufstieg
→ Induktion, Spezialfall einer schlichten Rekursion

Schlichte Rekursion

Berechnung nur auf einem Ast der Rekursion

- maximal ein rekursiver Aufruf pro Klausel
- zwei wichtige Fälle
 1. aufsteigende Rekursion, Induktion
 2. Endrekursion, absteigende Rekursion, repetitive Rekursion, tail recursion

aufsteigende Rekursion (Induktion)

- rekursiver Abstieg erfolgt nur, um den Induktionsanfang/Rekursionsabschluß zu finden
- alle wesentlichen Berechnungen erfolgen beim rekursiven Aufstieg
- z.B. Fakultät
- erlaubt oftmals richtungsunabhängige aber ineffiziente Implementierungen für arithmetische Probleme als generate-and-test-Verfahren.

Endrekursion (absteigende Rekursion)

- die gesamte Berechnung erfolgt beim rekursiven Abstieg
- das Ergebnis liegt am Rekursionsabschluss vor
- der rekursive Aufstieg dient nur der Übermittlung des Ergebnisses
- z.B. wenn Induktionsanfang / Rekursionsabschluss vor Berechnung noch unbekannt ist
- effiziente iterative Implementierung (ohne Verwendung des Stacks) ist möglich

Umwandlung in Endrekursion

Schlicht rekursive Berechnungsvorschriften, die aufsteigend sind (d.h. alle induktiven Programme), können immer in absteigende (endrekursive) Verfahren umgewandelt werden, wenn ein weiteres Argument zur Übergabe von Berechnungsergebnissen bereitgestellt wird (Augmentation).

Akkumulator

Zusätzliche Argumentposition, auf der das Berechnungsergebnis sukzessiv aufgebaut wird.

- Erinnerung: aufsteigend rekursive (induktive) Rekursion

$$\text{fak}(n, r) \leftarrow \begin{cases} r = 1 & \text{für } n = 0 \\ \text{fak}(n - 1, r_1), r = r_1 * n & \text{sonst} \end{cases}$$

```
% fak(+NatZahl,?Resultat)
% NatZahl und Resultat sind natuerliche Zahlen, so dass
% Resultat = fakultaet(NatZahl)
fak(0,1).                % Rekursionsabschluss
fak(N,FakN) :-           % Rekursionsschritt
    N > 0,                % nur fuer natuerliche Zahlen
    N1 is N - 1,          % Konsumtion des Eingabearguments
    fak(N1,FakN1),        % rekursiver Aufruf
    FakN is FakN1 * N.    % Aufbau des Resultats
```

Augmentation

- absteigend rekursive (endrekursive) Definition:

$$\text{fak}(n, r) \leftarrow \text{fak1}(n, 1, r)$$

$$\text{fak1}(n, a, r) \leftarrow \begin{cases} r = a & \text{für } n = 0 \\ \text{fak1}(n-1, n * a, r) & \text{sonst} \end{cases}$$

```
% fak_er(+NatZahl,?Resultat)
% NatZahl und Resultat sind natuerliche Zahlen, so dass
% Resultat = fakultaet(NatZahl) [ueber Endrekursion]

fak_er(N,FakN) :- fak1(N,1,FakN).
fak1(0,X,X).                               % Rekursionsabschluss
fak1(N,A,R) :-                               % Rekursionsschritt
    N > 0,                                   % nur fuer nat. Zahlen
    N1 is N - 1,                             % Konsumtion der Eingabe
    A1 is N * A,                             % Akkumulation des Resultats
    fak1(N1,A1,R).                          % rekursiver Aufruf
```

Vergleich

```
% fak(+NatZahl,?Resultat)
% aufsteigend rekursiv
```

```
fak(0,1).
fak(N,FakN) :-
    N > 0,
    N1 is N - 1,
    fak(N1,FakN1),
    FakN is FakN1 * N.
```

```
fak_er(+NatZahl,?Resultat)
endrekursiv
```

```
fak_er(N,FakN) :- fak1(N,1,FakN).
```

```
fak1(0,X,X).
fak1(N,A,R) :-
    N > 0,
    N1 is N - 1,
    A1 is N * A,
    fak1(N1,A1,R).
```

Augmentation

- Rekursionsschemata

fak(4,X)	fak(4,24)	fak1(4,1,X)	fak1(4,1,24)		
\	X is X1*4	/	\	X = X1	/
fak(3,X1)	fak(3,6)	fak1(3,4,X1)	fak1(3,4,24)		
\	X1 is X2*3	/	\	X1 = X2	/
fak(2,X2)	fak(2,2)	fak1(2,12,X2)	fak1(2,12,24)		
\	X2 is X3*2	/	\	X2 = X3	/
fak(1,X3)	fak(1,1)	fak1(1,24,X3)	fak1(1,24,24)		
(X3 is X4*1)	(X3 = X4)
fak(0,1)		fak1(0,24,24)			

Augmentation

- Rekursionsschemata

fak(4,X)	fak(4,24)	fak1(4,1,X)
\ X is X1*4 /		\ X = X1
fak(3,X1)	fak(3,6)	fak1(3,4,X1)
\ X1 is X2*3 /		\ X1 = X2
fak(2,X2)	fak(2,2)	fak1(2,12,X2)
\ X2 is X3*2 /		\ X2 = X3
fak(1,X3)	fak(1,1)	fak1(1,24,X3)
\ X3 is X4*1 /		\ X3 = X4
fak(0,1)		fak1(0,24,24)

Im rekursiven Aufstieg finden keine Berechnungen mehr statt.
Er kann entfallen!

Weitere Beispiele

- Umwandlung von PEANO-Zahlen
- größter gemeinsamer Teiler
- Russische Bauernmultiplikation

Umwandlung von PEANO-Zahlen

```
% int2peano(+nicht_negative_Integerzahl,?Peanozahl)

int2peano(0,0).          % Rekursionsabschluss
int2peano(N,s(P)):-
    N > 0,
    N1 is N - 1,         % Konsumtion
    int2peano(N1,P).     % Rekursionsschritt
```

- Konsumtion der Integer-Zahl

Umwandlung von PEANO-Zahlen

- Rekursionsschema

$$\begin{array}{ccc} \text{i2p}(4, X) & & \text{i2p}(4, s(s(s(s(0)))))) \\ \backslash & X = s(X1) & / \\ \text{i2p}(3, X1) & & \text{i2p}(3, s(s(s(0)))) \\ \backslash & X1 = s(X2) & / \\ \text{i2p}(2, X2) & & \text{i2p}(2, s(s(0))) \\ \backslash & X2 = s(X3) & / \\ \text{i2p}(1, X3) & & \text{i2p}(1, s(0)) \\ & \text{X3 = s(X4)} & \\ & \text{---i2p(0, 0)---} & \end{array}$$

Umwandlung von PEANO-Zahlen

- Rekursionsschema

```
i2p(4,X)
  \      X = s(X1)
i2p(3,X1)
  \      X1 = s(X2)
i2p(2,X2)
  \      X2 = s(X3)
i2p(1,X3)
  \      X3 = s(X4)
  \      i2p(0,0)
```

int2peano/2 ist endrekursiv: Beim rekursiven Aufstieg finden keine Berechnungen mehr statt. Er kann entfallen!

Umwandlung von PEANO-Zahlen

- induktive Implementierung
- Ziel: Richtungsunabhängigkeit wieder herstellen
- Ansatz: generate-and-test

```
% int2peano(?Integer,?Peano)  <-- Wunsch!  
int2peano(0,0).               % Rekursionsabschluss  
int2peano(N,s(P)) :-  
    int2peano(N1,P),          % Rekursionsschritt  
    N is N1 + 1.              % Erhöhen des Zählers
```

Umwandlung von PEANO-Zahlen

- Terminierungsproblem

```
?- int2peano(N,s(s(0))).
```

```
    N = 2 ;
```

```
    No
```

```
?- int2peano(2,P).
```

```
    P = s(s(0)) ;
```

```
    . . .
```

- hier (teilweise) Korrektur durch cut/0 möglich (aber eindeutiges Ergebnis auch bei vollständiger Unterspezifikation)

Umwandlung von PEANO-Zahlen

- Rekursionsschemata

$i2p(X, s(s(s(0)))) \quad i2p(3, s(s(s(0))))$

$\backslash \quad X \text{ is } X1 + 1 \quad /$

$i2p(X1, s(s(0))) \quad i2p(2, s(s(0)))$

$\backslash \quad X1 \text{ is } X2 + 1 \quad /$

$i2p(X2, s(0)) \quad i2p(1, s(0))$

$\underbrace{\quad X2 \text{ is } X3 + 1 \quad}_{i2p(0,0)}$

$i2p(3, X)$

$i2p(3, s(s(s(0))))$

$\backslash \quad 3 \text{ is } N1 + 1, X = s(X1) \quad /$

$i2p(N1, X1)$

$i2p(2, s(s(0)))$

$\backslash \quad N1 \text{ is } N2 + 1, X1 = s(X2) \quad /$

$i2p(N2, X2)$

$i2p(1, s(0))$

$\underbrace{\quad N2 \text{ is } N3 + 1, X2 = s(X3) \quad}_{i2p(0,0)}$

Größter gemeinsamer Teiler

- rekursive Definition (funktional)

$$\text{ggt}(x, y) = \begin{cases} x & \text{wenn } x = y \\ \text{ggt}(y, x) & \text{wenn } x < y \\ \text{ggt}(x - y, y) & \text{sonst} \end{cases}$$

- Rekursionsabschluss liefert gewünschtes Berechnungsergebnis
- induktive Berechnung nicht möglich
- Übertragung in eine relationale Definition (Augmentation)

$$\text{ggt}(x, y, g) = \begin{cases} g = x & \text{wenn } x = y \\ \text{ggt}(y, x, g) & \text{wenn } x < y \\ x_1 = x - y, \text{ggt}(x_1, y, g) & \text{sonst} \end{cases}$$

Größter gemeinsamer Teiler

- Übertragung in ein relationales Programm

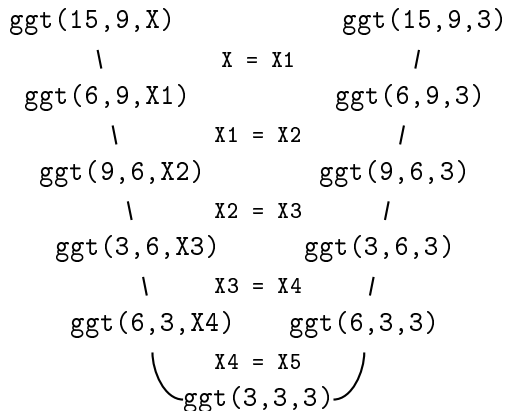
```
% ggt(+NatZahl1,+NatZahl2,-GGT)
% alle drei Argumente sind natuerliche Zahlen, mit GGT ist
% der groesste gemeinsame Teiler von NatZahl1 und NatZahl2
ggt(X,X,X).                                % Rekursionsabschluss
ggt(X,Y,Z) :- X<Y, ggt(Y,X,Z).             % Ordnen der Argumente
ggt(X,Y,Z) :- X>Y,                         % Rekursionsschritt
                X1 is X-Y,
                ggt(X1,Y,Z).
```

```
?- ggt(15,9,X).
   X=3.
```

```
?- ggt(2,3,X).
   X=1.
```

Größter gemeinsamer Teiler

- Rekursionsschema



Größter gemeinsamer Teiler

- Rekursionsschema

```
ggt(15, 9, X)
  |           X = X1
ggt(6, 9, X1)
  |           X1 = X2
ggt(9, 6, X2)
  |           X2 = X3
ggt(3, 6, X3)
  |           X3 = X4
ggt(6, 3, X4)
  |           X4 = X5
  |           ggt(3, 3, 3)
```

ggt/3 ist endrekursiv.

Russische Bauernmultiplikation

- rekursive Definition (funktional)

$$\text{rbm}(f_1, f_2) = \begin{cases} f_1 & \text{falls } f_2 = 1 \\ f_1 + \text{rbm}(f_1, f_2 - 1) & \text{falls ungeradzahlig}(f_2) \\ \text{rbm}(2f_1, f_2/2) & \text{sonst} \end{cases}$$

- Berechnung erfolgt sowohl beim rekursiven Abstieg (Fall 3), als auch beim rekursiven Aufstieg (Fall 2)
- schlichte Rekursion ist nicht möglich
- Übertragung in eine relationale Definition (Augmentation)

$$\text{rbm}(f_1, f_2, p) =$$

$$\begin{cases} p = f_1 & \text{falls } f_2 = 1 \\ f_{21} = f_2 - 1, \text{rbm}(f_1, f_{21}, p_1), p = p_1 + f_1 & \text{falls ungeradzahlig}(f_2) \\ f_{11} = f_1 * 2, f_{21} = f_2/2, \text{rbm}(f_{11}, f_{21}, p) & \text{sonst} \end{cases}$$

Russische Bauernmultiplikation

- Übertragung in ein relationales Programm

```
% rbm(+Faktor1,+Faktor2,-Produkt)
% die drei Argumente sind nat. Zahlen groesser Null
% mit Faktor1 * Faktor2 = Produkt

rbm(F1,1,F1).                                % Rekursionsabschluss
rbm(F1,F2,R) :- F2>1, odd(F2),               % Rekursionsschritt
                F21 is F2-1,                  % fuer unger. Faktor
                rbm(F1,F21,R1),
                R is R1+F1.
rbm(F1,F2,R) :- F2>1, even(F2),              % Rekursionsschritt
                F11 is F1*2,                  % fuer geraden Faktor
                F21 is F2/2,
                rbm(F11,F21,R).
```

Russische Bauernmultiplikation

- Hilfsprädikate

```
% odd(+NatZahl)
% NatZahl ist eine ungerade natuerliche Zahl
odd(X) :- X>0, 1 is X mod 2.
```

```
% even(+NatZahl)
% NatZahl ist eine gerade natuerliche Zahl
even(X) :- X>0, 0 is X mod 2.
```

```
?- odd(1).
true.
```

```
?- even(2).
true.
```

```
?- odd(2).
false.
```

```
?- even(1).
false.
```

```
?- odd(-1).
false.
```

```
?- even(0).
false.
```

Russische Bauernmultiplikation

- Testläufe

```
?- rbm(4,7,R).  
   R=28.
```

```
?- rbm(1,0,R).  
   false.
```

```
?- rbm(24,157,R).  
   R=3768.
```

```
?- rbm(-5,4,R).  
   R=-20.
```

```
?- rbm(1,1,R).  
   R=1.
```

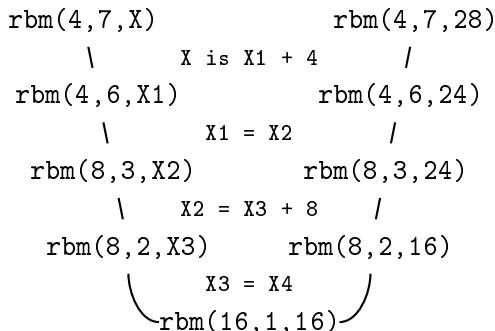
```
?- rbm(4,-5,R).  
   false.
```

```
?- rbm(0,1,R).  
   R=0.
```

```
?- rbm(X,5,20).  
   Instantiation Error  
?- rbm(5,X,20).  
   Instantiation Error
```

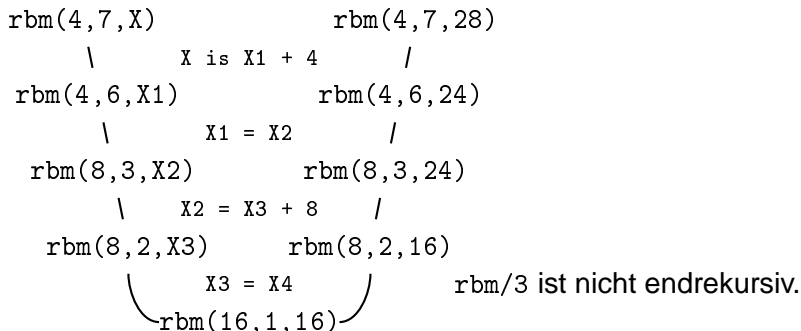
Russisches Bauernmultiplikation

- Rekursionsschema



Russisches Bauernmultiplikation

- Rekursionsschema

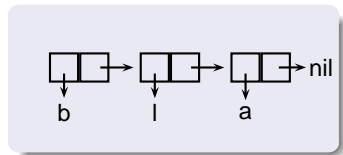


Testen eines Prädikats (4)

Beim systematischen Testen eines Prädikats müssen stets

- positive und negative Beispiele,
- alle Instanziierungsvarianten,
- alle wesentlichen Fallunterscheidungen
- wesentliche Verletzungen der Zusicherungen und
- wichtige Spezial- und Grenzfälle

berücksichtigt werden.



6. Verkettete Listen

Listennotation

Listenunifikation

Listenverarbeitung

Suchen und Sortieren

Memoization

Suchraumverwaltung

Bäume

Listen

- verkettete Listen sind die wichtigste rekursive Datenstruktur
- flexibel einsetzbar
 - Container für beliebige Datentypen (im Gegensatz zu Collections)
 - nicht längenbegrenzt
- sehr gute Grundlage für rekursive Prädikatsdefinitionen
→ spezielle Syntax zur bequemen Handhabung
- Vorgehen:
 - zuerst Beschränkung auf lineare Listen
 - später Erweiterung auf verzweigende Listen (Bäume)

Listennotation

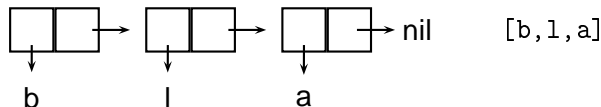
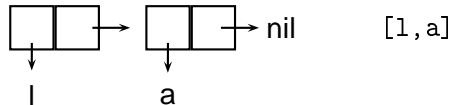
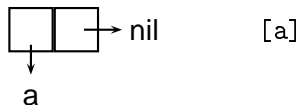
- Iterative Elementaufzählung: [a] [a, b] [a, b, c]
- leere Liste: []
- Rekursive Spezifikation:
 - Anfangselement: die leere Liste ist eine Liste
 - Nachfolgerbeziehung: ein *geordnetes Paar* aus einem Element und einer Liste ist eine Liste
 - Domänenabschluss: Nur die so gebildeten Objekte sind Listen
- Restlistenseparator/Listenkonstruktor:

[Listenkopf | Restliste]
- Gemischte Darstellung: [Kopf1, Kopf2, Kopf3 | Restliste]

Listennotation

- Implementation durch zweistellige Struktur

nil []



- vollständige Repräsentation der Zellenstruktur

$\lceil b \rceil \lceil l \rceil \lceil a \rceil \lceil \text{nil} \rceil$ ist synonym zu $\lceil b \ l \ a \rceil$

Listenunifikation

- Semantik (iterativ): Zwei Listen unifizieren, wenn ihre Elemente paarweise unifizieren. Die Unifikation schlägt fehl, wenn eine paarweise Zuordnung nicht möglich ist oder wenigstens ein Elementpaar nicht unifiziert.

?- [a, b, c] = [a, b, c] .
true.

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

?- [a, X, c] = [X, b, c] .
false.

?- [a, b, c] = [a, b] .
false.

Listenunifikation

- Semantik (rekursiv): Zwei Listen unifizieren, wenn die Listenköpfe und die Restlisten jeweils miteinander unifizieren. Die Unifikation schlägt fehl, wenn der Unifikationspartner keine Liste ist oder aber der Listenkopf bzw. die Restliste nicht unifizieren.

?- $[a, b, c, d] = [X | Y]$.
X=a, Y=[b, c, d] .

?- $[a, b, c, d] = [X, Y | _]$.
X=a, Y=b .

Listenunifikation

- Elementzugriff durch Unifikation (Dekomposition einer Liste)

?- [susi,hans,nina,paul]=[X|_].
X=susi.

?- [susi,hans,nina,paul]=[_,_ ,X|_].
X=nina.

- Konstruktion von Listen durch Unifikation (Komposition)

?- $L=[\text{susi}, \text{hans}, \text{nina}, \text{paul}]$, $EL=[\text{karl} | L]$.

$L = [\text{susi}, \text{hans}, \text{nina}, \text{paul}]$,

$EL = [\text{karl}, \text{susi}, \text{hans}, \text{nina}, \text{paul}]$.

?- $[X, Y | R]=[\text{susi}, \text{hans}, \text{nina}, \text{paul}]$, $EL=[X, Y, \text{karl} | R]$.

$X = \text{susi}, Y=\text{hans}, R=[\text{nina}, \text{paul}]$,

$EL = [\text{susi}, \text{hans}, \text{karl}, \text{nina}, \text{paul}]$.

- Unifikation rekursiv eingebetteter Listen

?- $[[a, b] \mid X] = [X, a, b] .$

$X = [a, b] .$

?- $[0, [P, Q], [a, 0]] = [[b, Q], [b, [c, P]], [a, [b, [R, b]]]]$

$0 = [b, [c, b]], P = b, Q = [c, b], R = c .$

Beispiel 1: Erstes Element einer Liste

- `first(?Element, ?Liste)`
- gewünschtes Verhalten:

1	<code>first(a, [a]).</code>	<code>true.</code>	4	<code>first(a, []).</code>	<code>false.</code>
2	<code>first(a, [a,b]).</code>	<code>true.</code>	5	<code>first(b, [a,b]).</code>	<code>false.</code>
3	<code>first(a, [a,b,c]).</code>	<code>true.</code>			

- rekursive Definition:

```
% first(?Element,?Liste)
% Element ist ein beliebiger Term und Liste ist eine
% Liste, so dass Element das erste Element der Liste ist
first(E, [E|_]).           % Der Kopf ist das
                           % erste Element einer Liste
```

Beispiel 1: Erstes Element einer Liste

- partiell unterspezifizierte Anfragen

```
?- first(X,[a,b,c]).           % first(-,+)  
    X=a.
```

```
?- first(a,X).               % first(+,-)  
    X=[a|X1].
```

Beispiel 1: Erstes Element einer Liste

- partiell unterspezifizierte Anfragen

```
?- first(X,X).                                % first(-,-)
   X = [X|_G202].
```

```
?- first(X,X), write(X).
   [**|_G202]
   X = [X|_G202].
```

```
?- first(X,X), X = [A|_], X = [[B|_|_|],
                               X = [[[C|_|_|]|_|_|].
   X = A, A = B, B = C, C = [C|_G202].
```

- vollständig unterspezifizierte Anfrage

```
?- first(X,Y).                                % first(-,-)
   X=X1, Y=[X1|Y1].
```

Beispiel 2: Element einer Liste

- `in_list(?Element,?Liste)`
- gewünschtes Verhalten:

1	<code>in_list(a,[a]).</code>	<code>true.</code>	5	<code>in_list(a,[]).</code>	<code>false.</code>
2	<code>in_list(a,[a,b]).</code>	<code>true.</code>	6	<code>in_list(d,[a,b,c]).</code>	<code>false.</code>
3	<code>in_list(b,[a,b,c]).</code>	<code>true.</code>			
4	<code>in_list(c,[a,b,c]).</code>	<code>true.</code>			

- rekursive Definition:

```
% in_list(?Element,?Liste)
% Element ist ein beliebiger Term und Liste eine Liste,
% so dass Element ein beliebiges Element der Liste ist

in_list(E,[E|_]).           % Der Kopf ist Element der Liste
in_list(E,[_|Rest]):-      % Ein Element der Restliste
    in_list(E,Rest).       %   ist auch Element der Liste
```

Beispiel 2: Element einer Liste

- partiell unterspezifizierte Anfragen:

```
?- in_list(X,[a,b,c]).  
    X=a ;  
    X=b ;  
    X=c .
```

```
?- in_list(a,X).  
    X=[a|X1] ;  
    X=[X2,a|X3] ;  
    X=[X2,X4,a|X5] .
```

- vollständig unterspezifizierte Anfrage:


```
?- in_list(X,Y).  
    X=X1, Y=[X1|Y1] ;  
    X=X1, Y=[X2,X1|Y2] ;  
    X=X1, Y=[X2,X3,X1|Y3] .
```

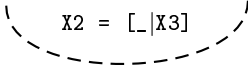
```
% in_list(-,-)
```

(oftmals) eingebautes Prädikat `member/2`

Beispiel 2: Element einer Liste

- Rekursionsschema: die Variablen werden auf jeder Rekursionsebene durch den Rekursionsabschluss instanziiert

$$\begin{array}{ccc} \text{il}(X, [a, b, c]) & & \text{il}(a, [a, b, c]) \\ \backslash & X = X1 & / \\ \text{il}(X1, [b, c]) & & \text{il}(b, [b, c]) \\ \backslash & X1 = X2 & / \\ \text{il}(X2, [c]) & & \text{il}(c, [c]) \end{array}$$


$$\begin{array}{ccc} \text{il}(a, X) & & \text{il}(a, [a|_]) \\ \backslash & X = [_|X1] & / \\ \text{il}(a, X1) & & \text{il}(a, [_|a|_]) \\ \backslash & X1 = [_|X2] & / \\ \text{il}(a, X2) & & \text{il}(a, [_|_|a|_]) \\ & X2 = [_|X3] & \end{array}$$


Beispiel 2: Element einer Liste

- Rekursionsschema: die Variablen werden auf jeder Rekursionsebene durch den Rekursionsabschluss instanziiert

```
il(X, [a, b, c])  
  \      X = X1  
  il(X1, [b, c])  
    \      X1 = X2  
    il(X2, [c])
```

```
il(a, X)  
  \      X = [_|X1]  
  il(a, X1)  
    \      X1 = [_|X2]  
    il(a, X2)  
      X2 = [_|X3]
```

Beispiel 3: Länge einer Liste

- `mylength(+Liste,-Laenge)`
- gewünschtes Verhalten:

1	<code>mylength([],0).</code>	<code>true.</code>	4	<code>mylength([a],0).</code>	<code>false.</code>
2	<code>mylength([a],1).</code>	<code>true.</code>	5	<code>mylength([a,b],1).</code>	<code>false.</code>
3	<code>mylength([a,b],2).</code>	<code>true.</code>			

- rekursive Definition:

```
% mylength(+Liste,-NatZahl)
% Liste ist eine Liste und NatZahl eine natuerliche Zahl
% mit NatZahl gleich der Anzahl der Elemente von Liste

mylength([ ],0).           % die leere Liste hat Laenge 0
mylength([_|Rest],N):-    % die Laenge einer Liste ist
    mylength(Rest,N1),    % die Laenge der Restliste
    N is N1+1.           % erhoeht um 1
```


Beispiel 3: Länge einer Liste

- partiell unterspezifizierte Anfragen:

```
?- mylength([a,b,c],X).           % mylength(+,-)
   X=3.
```

```
?- mylength(X,3).                % mylength(-,+)
   X=[X1,X2,X3] ;
   . . .
```

- vollständig unterspezifizierte Anfrage:

```
?- mylength(X,Y).               % mylength(-,-)
   Y=[ ], X=0 ;
   Y=[Y1], X=1 ;
   Y=[Y1,Y2], X=2 ;
   Y=[Y1,Y2,Y3], X=3 .
```

Beispiel 3: Länge einer Liste

- Rekursionsschema

$l([a,b,c],X)$ $l([a,b,c],3)$
 \ X is $X1 + 1$ /
 $l([b,c],X1)$ $l([b,c],2)$
 \ $X1$ is $X2 + 1$ /
 $l([c],X2)$ $l([c],1)$
 ($X2$ is $X3 + 1$)
 $l([],0)$

$l(X,3)$ $l([_,_,_],3)$
 \ $X = [_|X1], 3$ is $N1 + 1$ /
 $l(X1,N1)$ $l([_,_],2)$
 \ $X1 = [_|X2], N1$ is $N2 + 1$ /
 $l(X2,N2)$ $l([_],1)$
 ($X2 = [_|X3], N2$ is $N3 + 1$)
 $l([],0)$

Beispiel 4: Verketteten zweier Listen

- `app(?Liste1,?Liste2,?Gesamtliste)`
- gewünschtes Verhalten:

1	<code>app([],[],[]).</code>	yes
2	<code>app([],[a],[a]).</code>	yes
3	<code>app([],[a,b],[a,b]).</code>	yes
4	<code>app([a],[b,c],[a,b,c]).</code>	yes
5	<code>app([a,b],[c,d],[a,b,c,d]).</code>	yes
6	<code>app([a,b,c],[d,e],[a,b,c,d,e]).</code>	yes
7	<code>app([],[a],[]).</code>	no
8	<code>app([a,b],[c,d],[a,d]).</code>	no

4a	<code>app([a []],[b,c],[a [b,c]]).</code>	yes
5a	<code>app([a [b]],[c,d],[a [b,c,d]]).</code>	yes
6a	<code>app([a [b,c]],[d,e],[a [b,c,d,e]]).</code>	yes

Beispiel 4: Verketteten zweier Listen

- rekursive Definition:

```
% app(?Liste1,?Liste2,?Resultat)
% Liste1, Liste2 und Resultat sind Listen,
% so dass Resultat die Verkettung von Liste1
% und Liste2 ist

app([ ],L,L).                % [ ] ist neutrales Element
app([K|R1],L2,[K|VL]):- % Verk. von L2 mit der Restliste
    app(R1,L2,VL).          % ergibt die verkettete Restliste
```

Beispiel 4: Verketteten zweier Listen

- ein unterspezifiziertes Argument

```
?- app([a,b],[c,d,e],X).                                % app(+,+,-)  
    X=[a,b,c,d,e].
```

```
?- app(X,[c,d,e],[a,b,c,d,e]).                          % app(-,+,+)  
    X=[a,b].
```

```
?- app([a,b],X,[a,b,c,d,e]).                            % app(+,-,+)  
    X=[c,d,e].
```

Beispiel 4: Verketteten zweier Listen

- Rekursionsschema

$$\begin{array}{ccc} \text{app}([a, b, c], [d, e], X) & \text{app}([a, b, c], [d, e], [a, b, c, d, e]) \\ \backslash & X = [a|X1] & / \\ \text{app}([b, c], [d, e], X1) & \text{app}([b, c], [d, e], [b, c, d, e]) \\ \backslash & X1 = [b|X2] & / \\ \text{app}([c], [d, e], X2) & \text{app}([c], [d, e], [c, d, e]) \\ & X2 = [c|X3] & \\ & \text{app}([], [d, e], [d, e]) & \end{array}$$
$$\begin{array}{ccc} \text{app}(X, [d, e], [a, b, c, d, e]) & \text{app}([a, b, c], [d, e], [a, b, c, d, e]) \\ \backslash & X = [a|X1] & / \\ \text{app}(X1, [d, e], [b, c, d, e]) & \text{app}([b, c], [d, e], [b, c, d, e]) \\ \backslash & X1 = [b|X2] & / \\ \text{app}(X2, [d, e], [c, d, e]) & \text{app}([c], [d, e], [c, d, e]) \\ & X2 = [c|X3] & \\ & \text{app}([], [d, e], [d, e]) & \end{array}$$

Beispiel 4: Verketteten zweier Listen

- Rekursionsschema

$\text{app}([a, b, c], [d, e], X)$
 \
 $X = [a|X1]$
 $\text{app}([b, c], [d, e], X1)$
 \
 $X1 = [b|X2]$
 $\text{app}([c], [d, e], X2)$
 \
 $X2 = [c|X3]$
 $\text{app}([], [d, e], [d, e])$

$\text{app}(X, [d, e], [a, b, c, d, e])$ $\text{app}([a, b, c], [d, e], [a, b, c, d, e])$
 \
 $X = [a|X1]$ $/$
 $\text{app}(X1, [d, e], [b, c, d, e])$ $\text{app}([b, c], [d, e], [b, c, d, e])$
 \
 $X1 = [b|X2]$ $/$
 $\text{app}(X2, [d, e], [c, d, e])$ $\text{app}([c], [d, e], [c, d, e])$
 \
 $X2 = [c|X3]$
 $\text{app}([], [d, e], [d, e])$

Beispiel 4: Verketteten zweier Listen

- Rekursionsschema

```
app([a, b, c], [d, e], X)
    \           X = [a|X1]
app([b, c], [d, e], X1)
    \           X1 = [b|X2]
app([c], [d, e], X2)
    \           X2 = [c|X3]
    app([], [d, e], [d, e])
```

```
app(X, [d, e], [a, b, c, d, e])
    \           X = [a|X1]
app(X1, [d, e], [b, c, d, e])
    \           X1 = [b|X2]
app(X2, [d, e], [c, d, e])
    \           X2 = [c|X3]
    app([], [d, e], [d, e])
```


Beispiel 4: Verketteten zweier Listen

- Rekursionsschema

$$\begin{array}{ccc} \text{app}([a,b,c], X, [a,b,c,d,e]) & & \text{app}([a,b,c], [d,e], [a,b,c,d,e]) \\ \backslash & X = X1 & / \\ \text{app}([b,c], X1, [b,c,d,e]) & & \text{app}([b,c], [d,e], [b,c,d,e]) \\ \backslash & X1 = X2 & / \\ \text{app}([c], X2, [c,d,e]) & & \text{app}([c], [d,e], [c,d,e]) \\ & X2 = X3 & \\ & \text{app}([], [d,e], [d,e]) & \end{array}$$

Beispiel 4: Verketteten zweier Listen

- Rekursionsschema

```
app([a,b,c],X,[a,b,c,d,e])
      \                X = X1
app([b,c],X1,[b,c,d,e])
      \                X1 = X2
app([c],X2,[c,d,e])
      \                X2 = X3
      \
      \app([ ],[d,e],[d,e])
```

Beispiel 4: Verketteten zweier Listen

- zwei unterspezifizierte Argumente (1):

```
?- app(X,Y,[a,b,c,d,e]).                                % app(-,-,+)
   X=[ ],Y=[a,b,c,d,e] ;
   X=[a],Y=[b,c,d,e] ;
   X=[a,b],Y=[c,d,e] ;
   X=[a,b,c],Y=[d,e] ;
   X=[a,b,c,d],Y=[e] ;
   X=[a,b,c,d,e],Y=[ ] ;
no

?- app(X,[c,d,e],Y).                                     % app(-,+, -)
   X=[ ], Y=[c,d,e] ;
   X=[X1], Y=[X1,c,d,e] ;
   X=[X1,X2], Y=[X1,X2,c,d,e] ;
   X=[X1,X2,X3], Y=[X1,X2,X3,c,d,e]
yes
```

Beispiel 4: Verketteten zweier Listen

- Rekursionsschema

$$\begin{array}{l} \text{app}(X, Y, [a, b, c, d, e]) \qquad \text{app}([a, b, c], [d, e], [a, b, c, d, e]) \\ \quad \backslash \qquad \qquad X = [a|X1], Y = Y1 \qquad \qquad / \\ \text{app}(X1, Y1, [b, c, d, e]) \qquad \text{app}([b, c], [d, e], [b, c, d, e]) \\ \quad \backslash \qquad \qquad X1 = [b|X2], Y1 = Y2 \qquad \qquad / \\ \text{app}(X2, Y2, [c, d, e]) \qquad \text{app}([c], [d, e], [c, d, e]) \\ \qquad \qquad \qquad \qquad \qquad \qquad X2 = [c|X3], Y2 = Y3 \\ \qquad \qquad \qquad \qquad \qquad \qquad \text{app}([], [d, e], [d, e]) \end{array}$$
$$\begin{array}{l} \text{app}(X, [d, e], Y) \qquad \text{app}([_, _, _], [d, e], [_, _, _, d, e]) \\ \quad \backslash \qquad \qquad X = [_|X1], Y = [_|Y1] \qquad \qquad / \\ \text{app}(X1, [d, e], Y1) \qquad \text{app}([_, _], [d, e], [_, _, d, e]) \\ \quad \backslash \qquad \qquad X1 = [_|X2], Y1 = [_|Y2] \qquad \qquad / \\ \text{app}(X2, [d, e], Y2) \qquad \text{app}([_], [d, e], [_, d, e]) \\ \qquad \qquad \qquad \qquad \qquad \qquad X2 = [_|X3], Y2 = [_|Y3] \\ \qquad \qquad \qquad \qquad \qquad \qquad \text{app}([], [d, e], [d, e]) \end{array}$$

Beispiel 4: Verketteten zweier Listen

- Rekursionsschema

$\text{app}(X, Y, [a, b, c, d, e])$
 \
 $X = [a|X1], Y = Y1$
 $\text{app}(X1, Y1, [b, c, d, e])$
 \
 $X1 = [b|X2], Y1 = Y2$
 $\text{app}(X2, Y2, [c, d, e])$
 \
 $X2 = [c|X3], Y2 = Y3$
 $\text{app}([], [d, e], [d, e])$

$\text{app}(X, [d, e], Y)$ $\text{app}([_, _, _], [d, e], [_, _, _, d, e])$
 \
 $X = [_|X1], Y = [_|Y1]$ /
 $\text{app}(X1, [d, e], Y1)$ $\text{app}([_, _], [d, e], [_, _, d, e])$
 \
 $X1 = [_|X2], Y1 = [_|Y2]$ /
 $\text{app}(X2, [d, e], Y2)$ $\text{app}([_], [d, e], [_, d, e])$
 \
 $X2 = [_|X3], Y2 = [_|Y3]$
 $\text{app}([], [d, e], [d, e])$

Beispiel 4: Verketteten zweier Listen

- Rekursionsschema

```
app(X,Y,[a,b,c,d,e])  
  \      X = [a|X1], Y = Y1  
app(X1,Y1,[b,c,d,e])  
  \      X1 = [b|X2], Y1 = Y2  
app(X2,Y2,[c,d,e])  
      \      X2 = [c|X3], Y2 = Y3  
      app([ ],[d,e],[d,e])
```

```
app(X,[d,e],Y)  
  \      X = [_|X1], Y = [_|Y1]  
app(X1,[d,e],Y1)  
  \      X1 = [_|X2], Y1 = [_|Y2]  
app(X2,[d,e],Y2)  
      \      X2 = [_|X3], Y2 = [_|Y3]  
      app([ ],[d,e],[d,e])
```

Beispiel 4: Verketteten zweier Listen

- vollständig unterspezifizierter Aufruf

```
?- app(X,Y,Z).                                % app(-,-,-)
    X=[ ], Y=Y1, Z=Y1 ;
    X=[X1], Y=Y1, Z=[X1|Y1] ;
    X=[X1,X2], Y=Y1, Z=[X1,X2|Y1] ;
    X=[X1,X2,X3], Y=Y1, Z=[X1,X2,X3|Y1]
yes
```

(oftmals) eingebautes Prädikat

```
append(?Liste1,?Liste2,?Resultat)
```

Beispiel 5a: Suffix einer Liste

- rekursive Definition

```
% suffix(?Suffix,?Liste)
suffix(L,L).
suffix(S,[_|R]) :- suffix(S,R).
```

- partiell unterspezifizierte Anfragen

```
?- suffix(S,[a,b,c]).           % suffix(-,+)
   S = [a,b,c] ;
   S = [b,c] ;
   S = [c] ;
   S = [] .
```

```
?- suffix([a,b],L).           % suffix(+,-)
   L = [a,b] ;
   L = [_G298,a,b] ;
   L = [_G298,_G301,a, b] .
```


Beispiel 5b: Präfix einer Liste

- rekursive Definition

```
% prefix(?Prefix,?Liste)
prefix([],_).
prefix([E|P],[E|R]) :- prefix(P,R).
```

- unterspezifizierte Anfragen

```
?- prefix(P,[a,b,c]).           % prefix(-,+)
P = [] ;
P = [a] ;
P = [a,b] ;
P = [a,b,c] .
```

```
?- prefix([a,b],L).           % prefix(+,-)
L = [a,b|_G302] .
```

Beispiel 5c: Teilliste einer Liste

```
% sublist(?Subliste,?Liste)
sublist(S,L) :- prefix(S,L).
sublist(S,[_|R]) :- sublist(S,R).

?- sublist(S,[a,b,c]).           % sublist(-,+)
   S = [] ;
   S = [a] ;
   S = [a, b] ;
   S = [a, b, c] ;
   S = [] ;
   S = [b] ;
   S = [b, c] ;
   S = [] ;
   S = [c] ;
   S = [] .
```

Querbeziehungen in der Listenverarbeitung

- Basisprädikate zur Listenverarbeitung können wechselseitig auseinander definiert werden

```
member(E,L) :- sublist([E],L).
```

```
sublist(Sub,L) :- prefix(Pre,L), suffix(Sub,Pre).
```

```
sublist(Sub,L) :- suffix(Suf,L), prefix(Sub,Suf).
```

```
prefix(P,L) :- append(P,_,L).
```

```
suffix(S,L) :- append(_,S,L).
```

```
sublist(Sub,L) :- append(_,Suf,L), append(Sub,_,Suf).
```

```
sublist(Sub,L) :- append(Pre,_,L), append(_,Sub,Pre).
```

```
sublist(Sub,L) :- prefix(Pre,L), suffix(Suf,L),  
    append(Pre,Sub,L1), append(L1,Suf,L).
```

...

Beispiel 6: Umdrehen einer Liste

- `reverse(?Liste1,?Liste2)`
- gewünschtes Verhalten:

1	<code>reverse([],[])</code>	yes
2	<code>reverse([a],[a])</code>	yes
3	<code>reverse([a,b],[b,a])</code>	yes
4	<code>reverse([a,b,c],[c,b,a])</code>	yes

- Direkte, längenunabhängige Implementierung?

Beispiel 6: Umdrehen einer Liste

- Repräsentation von Teilergebnissen:

1	<code>rev1([a,b,c],[])</code>	yes
2	<code>rev1([b,c],[a])</code>	yes
3	<code>rev1([c],[b,a])</code>	yes
4	<code>rev1([],???)</code>	yes

- Augmentation: zusätzliches Argument für Endergebnis:

1	<code>rev2([a,b,c],[],_)</code>	yes
2	<code>rev2([b,c],[a],_)</code>	yes
3	<code>rev2([c],[b,a],_)</code>	yes
4	<code>rev2([],[c,b,a],[c,b,a])</code>	yes

Beispiel 6: Umdrehen einer Liste

- Rekursive Prädikatsdefinition:

```
rev2([ ],L,L).  
rev2([H|T],A,R):-rev2(T,[H|A],R).
```

- Einkleiden zur Unterdrückung des Hilfsarguments

```
% reverse(?Liste1,?Liste2)  
% Liste1 und Liste2 sind Listen, so dass sie jeweils  
% die Elemente der anderen Liste in umgekehrter  
% Reihenfolge enthalten  
  
reverse(L,R):-  
    rev2(L,[ ],R).
```

Beispiel 6: Umdrehen einer Liste

- Rekursionsschema

rev2([a,b,c],[],X)		rev2([a,b,c],[],[c,b,a])
\	X = X1	/
rev2([b,c],[a],X1)		rev2([b,c],[a],[c,b,a])
\	X1 = X2	/
rev2([c],[b,a],X2)		rev([c],[b,a],[c,b,a])
└────────── X2 = X3 ─────────┘		
rev2([],[c,b,a],[c,b,a])		

Beispiel 6: Umdrehen einer Liste

- Rekursionsschema

```
rev2([a,b,c],[ ],X)
    \
    rev2([b,c],[a],X1)
        \
        rev2([c],[b,a],X2)
            \
            rev2([ ],[c,b,a],[c,b,a])
```

X = X1
X1 = X2
X2 = X3

Beispiele 7a: Löschen von Elementen

- alle Vorkommen des Elements

```
% delete_element(?Element,?Liste,?RedListe)
delete_element(_,[ ],[ ]).
delete_element(E,[E|R],RL) :-
    delete_element(E,R,RL).
delete_element(E,[X|R],[X|RL]) :-
    X\=E, delete_element(E,R,RL).

% sinnvolle Verwendung:
?- delete_element(a,[a,f,d,w,a,g,t,s,a],L).
   L = [f, d, w, g, t, s].
```

Beispiele 7a: Löschen von Elementen

- alle Vorkommen des Elements

```
% delete_element(?Element,?Liste,?RedListe)
```

```
delete_element(_,[ ],[ ]).
```

```
delete_element(E,[E|R],RL) :-
```

```
    delete_element(E,R,RL).
```

```
delete_element(E,[X|R],[X|RL]) :-
```

```
    X\=E, delete_element(E,R,RL).
```

```
% sinnvolle Verwendung:
```

```
?- delete_element(a,[a,f,d,w,a,g,t,s,a],L).
```

```
    L = [f, d, w, g, t, s].
```

→ delete/3 (deprecated)

Beispiel 7b: Löschen eines Elements

- Entfernen nur eines Vorkommens des gegebenen Elementes

```
% select_element(?Element,?Liste,?RedListe)
select_element(E,[E|R],R).
select_element(E,[X|R],[X|RL]) :-
    select_element(E,R,RL).
```

% sinnvolle Verwendungen

```
?- select_element(a,[a,f,d,w,a,g,t,s,a],L).
    L = [f, d, w, a, g, t, s, a] ;
    L = [a, f, d, w, g, t, s, a] ;
    L = [a, f, d, w, a, g, t, s].
```

```
?- select_element(E,[a,b,c],L).
    E = a, L = [b, c] ;
    E = b, L = [a, c] ;
    E = c, L = [a, b].
```

Beispiel 7b: Löschen eines Elements

- Entfernen nur eines Vorkommens des gegebenen Elementes

```
% select_element(?Element,?Liste,?RedListe)
select_element(E,[E|R],R).
select_element(E,[X|R],[X|RL]) :-
    select_element(E,R,RL).
```

% sinnvolle Verwendungen

```
?- select_element(a,[a,f,d,w,a,g,t,s,a],L).
    L = [f, d, w, a, g, t, s, a] ;
    L = [a, f, d, w, g, t, s, a] ;
    L = [a, f, d, w, a, g, t, s].
```

```
?- select_element(E,[a,b,c],L).
    E = a, L = [b, c] ;
    E = b, L = [a, c] ;
    E = c, L = [a, b].
```

→ select/3

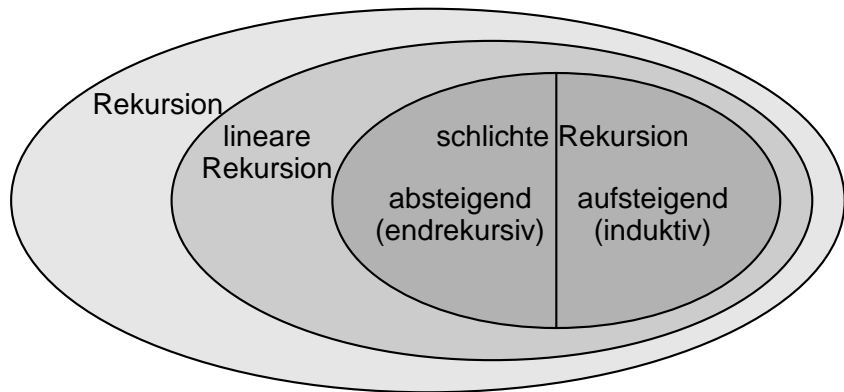
Beispiel 8: Einebnen einer Liste

- Erzeugen einer linearen Liste aus einer rekursiv geschachtelten Liste

```
% flatten(+geschachtelte Liste,?lineare_Liste)
flatten([],[]).
flatten([K|R],[K|RF]) :-
    atomic(K),
    flatten(R,RF).
flatten([K|R],F) :-
    is_list(K),
    flatten(K,KF),
    flatten(R,RF),
    append(KF,RF,F).
```

- verzweigende Rekursion: mehrere rekursive Aufrufe in einer Klausel

Überblick Rekursion



Anwendungsbereich 1: Suchen und Sortieren

- Suchen in einer sortierten Liste
- Sortieren durch Einfügen in eine sortierte Liste
- Sortieren durch Auswahl des minimalen Elements
- Sortieren durch rekursives Zerlegen einer Liste (Quicksort)
- Sortieren durch Aufbau eines (sortierten) Baumes

Suche in einer sortierten Liste

```
% member_sort(+Element,+SortierteListe)
member_sort(E,[E|_]).
member_sort(E,[X|R]) :-
    E @> X,
    member_sort(E,R).
```


Sortieren durch Permutation

```
% permutation(+Liste,?Permutierte Liste)
permutation([],[]).
permutation(L,[E|R]) :- select_element(E,L,L1),
    permutation(L1,R).
```

```
?- permutation([a,b,c],L).
    L = [a, b, c] ;
    L = [a, c, b] ;
    L = [b, a, c] ;
    L = [b, c, a] ;
    L = [c, a, b] ;
    L = [c, b, a].
```

Sortieren durch Einfügen

- Einfügen in eine sortierte Liste

```
% sort_e(+Liste,?SortierteListe)
sort_e([ ],[ ]).
sort_e([E|L],SL) :-
    sort_e(L,SL1),
    insert_l(E,SL1,SL).
```

Sortieren durch Einfügen

- Einfügen eines Elements

```
% insert_1(+Element,+SortierteListe,?ErwListe)
insert_1(E,[ ],[E]).
insert_1(E,[X|R],[X|RL]) :-
    E@>X,
    insert_1(E,R,RL).
insert_1(E,[X|R],[E,X|R]) :-
    E@=<X.
```

Sortieren durch Auswahl des minimalen Elements

- Entferne das kleinste Element und setze es an den Listenanfang

```
% sort_a(+Liste,?SortierteListe)
```

```
sort_a([E],[E]).
```

```
sort_a(L,[M|R]):-
```

```
    select_minimum(M,L,Rest),
```

```
    sort_a(Rest,R).
```

```
% select_minimum(?Minimum,+Liste,?Restliste)
```

```
select_minimum(E,L,R) :-
```

```
    minimum(E,L),
```

```
    select_element(E,L,R).
```

Sortieren durch Auswahl des minimalen Elements

- Minimales Element einer Liste

```
% minimum(?MinimalesElement,+Liste)
minimum(M, [M] ).
minimum(M, [X|R] ) :-
    minimum(M,R),
    M@<X.
minimum(X, [X|R] ) :-
    minimum(M,R),
    M@>=X.
```

Sortieren durch Auswahl des minimalen Elements

- Minimales Element einer Liste

```
% minimum(?MinimalesElement,+Liste)
minimum(M, [M] ).
minimum(M, [X|R] ) :-
    minimum(M,R),
    M@<X.
minimum(X, [X|R] ) :-
    minimum(M,R),
    M@>=X.
```

→ min_member/2

Quicksort

- Sortieren durch rekursive Zerlegung in Teilprobleme

```
% sort_q(+Liste,?SortierteListe)
sort_q([],[]).
sort_q([E|R],SL) :-
    split(R,E,Vorn,Hinten),
    sort_q(Vorn,VS),
    sort_q(Hinten,HS),
    append(VS,[E|HS],SL).
```

Quicksort

- Zerlegen einer Liste

```
%split(+Liste,+MittleresElement,  
%      ?VordereElemente,?HintereElemente)  
split([ ],_,[ ],[ ]).  
split([E|R],M,[E|VL],HL) :-  
    E@=<M, split(R,M,VL,HL).  
split([E|R],M,VL,[E|HL]) :-  
    E@>M, split(R,M,VL,HL).  
  
?- split([c,a,e,b],d,V,H).  
    V = [c, a, b], H = [e].
```


Sortieren durch Konstruktion eines Baumes

- Umwandeln einer Liste in einen sortierten Baum und Rückumwandeln in eine (sortierte) Liste

```
% sort_t(+Liste,?SortierteListe)
```

```
sort_t(L,SL) :-
```

```
    list2tree(L,B),
```

```
    tree2list(B,SL).
```

Sortieren durch Konstruktion eines Baumes

- Umwandeln der Liste in einen Baum

```
% list2tree(+Liste,?Baum)
list2tree([ ],end).
list2tree([E|R],t(E,VB,HB)) :-
    split(R,E,VL,HL),
    list2tree(VL,VB),
    list2tree(HL,HB).

?- list2tree([d,c,a,e,b],B).
   B = t(d, t(c, t(a, end, t(b, end, end)),
          end), t(e, end, end)).
```

Sortieren durch Konstruktion eines Baumes

- Umwandeln der Liste beim rekursiven Aufstieg

```
% list2tree(+Liste,?Baum)
list2tree([E],t(E,end,end)).
list2tree([E|R],t(S,VB,HB)) :-
    list2tree(R,t(S1,VB1,HB1)),
    intree(E,t(S1,VB1,HB1),t(S,VB,HB)).

?- list2tree([d,c,a,e,b],B).
   B = t(b, t(a, end, end), t(e, t(c, end,
        t(d, end, end)), end)).
```

Sortieren durch Konstruktion eines Baumes

- Einfügen eines Elementes in einen (sortierten) Baum

```
% intree(+Element,+BaumAlt,?BaumNeu)
intree(E,end,t(E,end,end)).
intree(E,t(S,VB,HB),t(S,VBN,HB)) :-
    E@=<S, intree(E,VB,VBN).
intree(E,t(S,VB,HB),t(S,VB,HBN)) :-
    E@>S, intree(E,HB,HBN).
```

Sortieren durch Konstruktion eines Baumes

- Beispielaufruf

```
?- intree(f,t(d, t(c, t(a, end, t(b, end, end)),  
              end), t(e, end, end)),T).
```

```
T = t(d, t(c, t(a, end, t(b, end, end)),  
        end), t(e, end, t(f, end, end))).
```

Sortieren durch Konstruktion eines Baumes

- Umwandeln des Baumes in eine Liste

```
% tree2list(+Baum,?Liste)
tree2list(end,[ ]).
tree2list(t(E,VB,HB),L) :-
    tree2list(VB,VL),
    tree2list(HB,HL),
    append(VL,[E|HL],L).
```

Sortieren durch Konstruktion eines Baumes

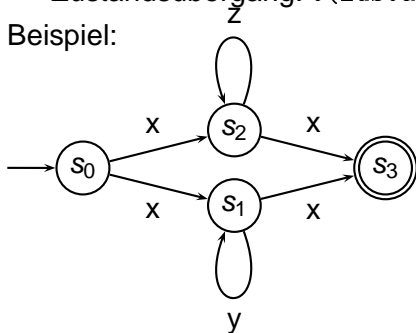
```
?- tree2list(t(d, t(c, t(a, end, t(b, end, end)),
    end), t(e, end, t(f, end , end)),L).
L = [a, b, c, d, e, f].
```

Anwendungsbereich 2: Memoization

- Memoization: Buchführung über die bisher besuchten Suchzustände
 - Ausgabe der Zustandsfolge
 - z.B. Endlicher Automat
 - Überwachung von Zyklen
 - z.B. Suche in einem Labyrinth

Anwendungsbeispiel: Endlicher Automat

- Repräsentation eines Endlichen Automaten:
 - Startzustand: $a(zustand)$
 - Zielzustand: $e(zustand)$
 - Zustandsübergang: $t(zustand-alt, symbol, zustand-neu)$
- Beispiel: Regulärer Ausdruck: $x(y^* | z^*)x$



Anwendungsbeispiel: Endlicher Automat

- Repräsentation des Beispielautomaten:

a(s0).	t(s0,x,s1).
	t(s0,x,s2).
e(s3).	t(s1,x,s3).
	t(s2,x,s3).
	t(s1,y,s1).
	t(s2,z,s2).

- Repräsentation der Zeichenfolge als Liste:

[x, z, z, z, x]

Anwendungsbeispiel: Endlicher Automat

```
% generate(?Wort)
% Wort ist eine Liste von Symbolen, so dass
% Wort Element der durch den endlichen
% Automaten definierten Sprache ist

generate(Wort):-
    a(Start),
    gen(Wort,Start,Ziel),
    e(Ziel).

gen([ ],Start,Start).
gen([Kopf|Rest],Start,Ziel):-
    t(Start,Kopf,Zwischenzustand),
    gen(Rest,Zwischenzustand,Ziel).
```

Anwendungsbeispiel: Endlicher Automat

```
?- generate([x,y,x]).  
    true.  
?- generate([x,y]).  
    false.  
?- generate([y,z]).  
    false.  
?- generate(X).  
    X=[x,x] ;  
    X=[x,y,x] ;  
    X=[x,y,y,x] ;  
    X=[x,y,y,y,x] .
```

Anwendungsbeispiel: Zyklenvermeidung im Labyrinth

- Repräsentation eines Labyrinths

```
% weg(Ort_a, Ort_b)
c(a,b).
c(b,c).
c(a,d).
c(b,d).
c(c,d).
weg(A,B) :- c(A,B).
weg(A,B) :- c(B,A).
weg(A,C) :- c(A,B), weg(B,C).
weg(A,C) :- c(B,A), weg(B,C).
```

Anwendungsbeispiel: Zyklenvermeidung im Labyrinth

?- weg(d,X) .

X = a ;

X = b ;

X = c ;

X = b ;

X = d ;

...

X = a ;

X = b ;

X = c ;

X = b ;

X = d ;

...

Anwendungsbeispiel: Zyklenvermeidung im Labyrinth

- Protokollierung der (virtuell) zurückgelegten Wegstrecke in einem Akkumulator (3. Argument)
- Initialzustand ist der Ausgangsort
- Abbruch bei erneutem Auftreten eines bereits besuchten Ortes

Anwendungsbeispiel: Zyklenvermeidung im Labyrinth

```
% weg_oz(?Start,?Ziel)
% zyklensfreie Verbindung
weg_oz(A,B) :- w_oz(A,B,[A]).
w_oz(A,B,W) :-
    c(A,B), \+ member(B,W).
w_oz(A,B,W) :-
    c(B,A), \+ member(B,W).
w_oz(A,C,W) :-
    c(A,B), \+ member(B,W), w_oz(B,C,[B|W]).
w_oz(A,C,W) :-
    c(B,A), \+ member(B,W), w_oz(B,C,[B|W]).
```


Anwendungsbeispiel: Zyklenvermeidung im Labyrinth

```
?- weg_oz(d,X) .
```

```
  X = a ;
```

```
  X = b ;
```

```
  X = c ;
```

```
  X = b ;
```

```
  X = c ;
```

```
  X = c ;
```

```
  X = a ;
```

```
  X = b ;
```

```
  X = a .
```

Anwendungsbeispiel: Zyklenvermeidung im Labyrinth

- Ausgabe der zurückgelegten Wegstrecke

```
% weg_oz(?Start,?Ziel,?BishWeg,?GesWeg)
% zyklensfreie Verbindung
weg_oz(A,B,G) :- w_oz(A,B,[A],R), reverse(R,G).
w_oz(A,B,W,[B|W]) :-
    c(A,B), \+ member(B,W).
w_oz(A,B,W,[B|W]) :-
    c(B,A), \+ member(B,W).
w_oz(A,C,W,GW) :-
    c(A,B), \+ member(B,W), w_oz(B,C,[B|W],GW).
w_oz(A,C,W,GW) :-
    c(B,A), \+ member(B,W), w_oz(B,C,[B|W],GW).
```

Anwendungsbeispiel: Zyklenvermeidung im Labyrinth

```
?- weg_oz(d,X,G) .  
    X = a, G = [d, a] ;  
    X = b, G = [d, b] ;  
    X = c, G = [d, c] ;  
    X = b, G = [d, a, b] ;  
    X = c, G = [d, a, b, c] ;  
    X = c, G = [d, b, c] ;  
    X = a, G = [d, b, a] ;  
    X = b, G = [d, c, b] ;  
    X = a, G = [d, c, b, a] .
```

Anwendungsbereich 3: Suchraumverwaltung

- Prolog realisiert standardmäßig eine Tiefe-zuerst-Suche
- übernimmt das Programm selbst die Verwaltung der Suchraumzustände, können auch alternative Suchstrategien realisiert werden

Anwendungsbereich 3: Suchraumverwaltung

- Beispiel: endlicher Automat
 - Verwalten einer Agenda auf der ersten Argumentstelle
 - Zustand des Automaten
 - Bisher erzeugtes Wort (rückwärts)
- z.B. [s1, [y, y, x]]
- Abarbeiten der Agenda vom Listenanfang
 - Hinzufügen neuer Agendaelemente
 - am Listenanfang → Kellerspeicher (Stack)
 - am Listenende → Warteschlange (Queue)

Anwendungsbereich 3: Suchraumverwaltung

- Erzeugen eines Wortes

```
% generate(?Wort)
generate(Word) :-
    findall([A,[ ]],a(A),Agenda),
    gen(Agenda,WordR),
    reverse(WordR,Word).
```

Anwendungsbereich 3: Suchraumverwaltung

- Abarbeiten der Agenda

```
% gen(+Agenda,?BisherigesWort)
gen([[End,PWord] | _],PWord) :- e(End).
gen([[Z,PWord] | AgendaR],Word) :-
    findall([ZNext,[E|PWord]],
            t(Z,E,ZNext),
            NewItems),
% append(NewItems,AgendaR,AgendaNew), % stack
append(AgendaR,NewItems,AgendaNew), % queue
gen(AgendaNew,Word).
```

Anwendungsbereich 3: Suchraumverwaltung

- Aufruf mit Kellerspeicher

```
?- generate(X) .  
    X = [x, x] ;  
    X = [x, y, x] ;  
    X = [x, y, y, x] ;  
    X = [x, y, y, y, x] ;  
    X = [x, y, y, y, y, x] ;  
    X = [x, y, y, y, y, y, x] .
```


Anwendungsbereich 3: Suchraumverwaltung

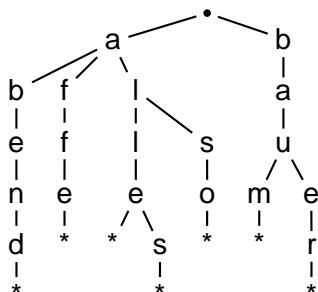
- Aufruf mit Warteschlange

```
?- generate(X).  
  X = [x, x] ;  
  X = [x, x] ;  
  X = [x, y, x] ;  
  X = [x, z, x] ;  
  X = [x, y, y, x] ;  
  X = [x, z, z, x] ;  
  X = [x, y, y, y, x] ;  
  X = [x, z, z, z, x] .
```

Bäume

- SE I

```
[b, [a, [u, [m, []],
            [e, [r, []]]]]]] ).
```



- Test: Ist Wort (gegeben als Liste) im Trie enthalten?

```
% word(?Word,?Trie).
```

```
word([], [[] | _]).
```

```
word([C|RW], [[C|RT] | _]) :- word(RW, RT).
```

```
word(W, [_|Alt]) :- word(W, Alt).
```

```
?- trie(T), word([a, l, s, o], T).      % word(+, +)
```

```
    T = [...].
```

```
?- trie(T), word([a, b, e, r], T).      % word(+, +)
```

```
    false.
```

Bäume

```
?- trie(T),word(W,T).                % word(-,+)
```

```
    T = [...]
```

```
    W = [a, b, e, n, d] ;
```

```
    T = [...]
```

```
    W = [a, f, f, e] ;
```

```
    T = [...]
```

```
    W = [a, l, l, e] ;
```

```
    ...
```

```
    T = [...]
```

```
    W = [b, a, u, e, r].
```

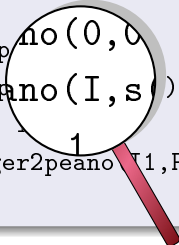
```
?- word([a,l,s,o],T).                % word(+,-)
```

```
    T = [[a, [l, [s, [o, []|...]|_G259]|_G253]|_G247]|_G241] ;
```

```
    T = [[a, [l, [s, [o, _G264|...]|_G259]|_G253]|_G247]|_G241]
```

```
    true.
```

Extra- und Metalogische Prädikate



```
integer2peano(0,0).  
integer2peano(I,s(I)) :-  
    I1 is I-1,  
    integer2peano(I1,P).
```

7. Extra- und Metalogische Prädikate

- Typen und Typkonversion
- Kontrollierte Instanziierung
- Suchraummanipulation
- Extralogische Prädikate

Extra- und Metalogische Prädikate

Metalogische Prädikate

- Typinspektion
- Typkonversion
- Typkonstruktion
- Kontrollierte Instantiierung
- Suchraummanipulation
- (Trace und Debug)

Extralogische Prädikate

- (Input/Output)
- Datenbankmanipulation
- globale Variable
- (Interface zum Betriebssystem)

Atomare Basistypen

<code>integer/1</code>	Ganze Zahlen
<code>float/1</code>	Gleitkommazahlen
<code>atom/1</code>	Atome (Namen)
<code>string/1</code>	SWI-String

Abgeleitete atomare Typen

```
number(X) :- integer(X).  
number(X) :- float(X).  
atomic(X) :- string(X).  
atomic(X) :- atom(X).  
atomic(X) :- number(X).
```


- Beispiel: Terminierungssicheres ggt/3

```
% ggt(+NatZahl1,+NatZahl2,?GGT)
ggt(X,Y,Z) :- integer(X), integer(Y),
    X>0, Y>0, ggt1(X,Y,Z).
ggt1(X,X,X).
ggt1(X,Y,Z) :- X<Y, ggt1(Y,X,Z).
ggt1(X,Y,Z) :-
    X>Y, X1 is X-Y, ggt1(X1,Y,Z).

?- ggt(1.5,2.1,X).
false.
```

Zeichenketten

- Datentyp `string`
 - im originalen Edinburgh-Prolog als Liste von ASCII-Werten
 - Konvertierung direkt beim Einlesen `read/1`
 - logisch "sauber"
 - aber ineffizient
- daher in ISO-Norm: separater Datentyp `String`

```
?- X="string".  
   X = "string".
```

```
?- X="string", string(X).  
   X = "string".
```

Zeichenketten

- ab SWI-Prolog 7.0: Strings als Listen von ASCII-Codes mit Backquotes

```
?- X = "string", string(X).  
X = "string".
```

```
?- X = 'string', string(X).  
false.
```

```
?- X = 'string', is_list(X).  
X = [115, 116, 114, 105, 110, 103].
```

Zeichenketten

- drei Möglichkeiten zur Repräsentation von Text-Daten
 - als Atom: 'Atom'
wenn als Identifier verwendet
 - als Listen von ASCII-codes: 'a+b'
wenn der Text strukturell analysiert werden soll, z.B.
Programmiersprachenausdrücke
 - als Zeichenketten (String): "String"
wenn weder Identifier noch strukturelle Analyse notwendig

Strukturen

- Strukturtypen

`compound(+Term)`

`compound(X) :-`

`\+ atomic(X)`

`is_list(+List)`

oberste Ebene ist gepunktetes
Paar

`proper_list(+List)`

rekursiver Nachweis

`is_set(+List)`

wie `proper_list`, aber keine Dupli-
kate

Typkonversion

- arithmetische Funktionen

float

integer

- Atom - Zeichenkette - Struktur

```
atom_codes(?Atom,?List_of_char_codes)
name(?AtomOrInteger,?List_of_char_codes)
atom_char(?Atom,?ASCII_Value)
int_to_atom(+Integer,+Base,-Atom)
term_to_atom(?Term,?Atom)
string_to_atom(?String,?Atom)
string_codes(?String,?List_of_char_codes)
    äquivalent zu: string_to_list/2
```

...

- Strings

```
?- string_to_list("ABC",L), atom_codes(A,L).  
L = [65, 66, 67],  
A = 'ABC'.
```

```
?- term_to_atom(p(1,2),A), atom_codes(A,L),  
   string_to_atom(S,A).  
A = 'p(1,2)',  
L = [112, 40, 49, 44, 50, 41],  
S = "p(1,2)".
```


Typspezifische Accessoren

- Atome

atom_length(+Atom, -Length)

string_length(+String, -Length)

substring(+String, +Start, +Len, -Substr)

- Listen

last/2

member/2

nth0/3, nth1/3

length/2

...

- Atome

`atom_chars(?Atom,?ListOfChars)`

`number_chars(?Number,?ListOfDigits)`

`concat(?Atom1,?Atom2,?Atom3)`

`concat(+ListOfAtoms,-Atom)`

- Listen

```
% ?Term =.. ?List
```

```
?- a(b,c) =.. X.
```

```
    X = [a,b,c].
```

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

```
    X = a(b,c).
```

Variableninspektion

```
% var(+Term)
?- var(X).
   X = _G123.

?- var(a).
   false.
```

Variableninspektion

```
% nonvar(+Term)
?- nonvar(X).
   false.

?- nonvar(a).
   true.

?- nonvar(a(X)).
   X = _G215.
```

Variableninspektion

```
% ground(+Term)
```

```
?- ground(X).  
    false.
```

```
?- ground(a(X)).  
    false.
```

```
?- ground(a(b)).  
    true.
```

Variableninspektion

- Beispiel: (partiell) relationale Addition
 - wenigstens zwei Argumente müssen instanziiert sein

```
% add(?Zahl1, ?Zahl2, ?Sum)
```

```
add(X,Y,Z) :-  
    nonvar(X), nonvar(Y), Z is X + Y.
```

```
add(X,Y,Z) :-  
    nonvar(X), nonvar(Z), Y is Z - X.
```

```
add(X,Y,Z) :-  
    nonvar(Y), nonvar(Z), X is Z - Y.
```

Variableninspektion

- Beispiel: (partiell) relationale Addition

```
?- add(3,4,X) .  
    X=7 .
```

```
?- add(1,X,4) .  
    X=3 .
```

```
?- add(X,Y,7) .  
    false .
```


Instanziierungsfreier Vergleich

- Vergleich variablenhaltiger Terme ohne Instanziierung der freien Variablen
 - strukturelle Identität
 - strukturelle Gleichheit

strukturelle Identität

- gleicher Typ, gleicher Funktor, gleiche Stelligkeit, gleiche Argumente
- Identität von Variablen: gleicher Name oder vorherige Unifikation

<code>a == a</code>	<code>true</code>	<code>a(X) == a(X)</code>	<code>true</code>
<code>a == A</code>	<code>false</code>	<code>a(X) == a(Y)</code>	<code>false</code>
<code>a(b) == a(b)</code>	<code>true</code>	<code>X = Y, a(X) == a(Y)</code>	<code>true</code>

Instanziierungsfreier Vergleich

strukturelle Gleichheit

- Identitätstest, aber Variable sind auch dann gleich, wenn sich bei konsistenter Umbenennung gleiche Koreferenzbeziehungen ergeben

$a =@= A$	false	$x(A,A) =@= x(B,C)$	false
$A =@= B$	true	$x(A,A) =@= x(B,B)$	true
		$x(A,B) =@= x(C,D)$	true

- Strukturelle Identität
 - ⊂ Strukturelle Gleichheit
 - ⊂ Unifizierbarkeit

Unifikationsfreie Strukturerzeugung

- `free_variables(+Term, -ListOfFreeVariables):`
Erzeugt eine Liste aller freien Variablen in Term
- `copy_term(+In, -Out):`
Erzeugt einen strukturgleichen Term

Erzeugen uninstanciierter Terme

```
% functor(?Term,?Functor,?Arity)
?- functor(a(b),X,Y).
   X = a, Y = 1.

?- functor(X,a,2).
   X = a(_G123, _G125).
```

Instantiieren von einzelnen Argumenten

```
% arg(?ArgNo,?Term,?Value)
```

```
?- arg(1,a(b,c),X).
```

```
    X = b.
```

```
?- functor(X,a,2),arg(1,X,b),arg(2,X,c).
```

```
    X = a(b, c).
```

Suchraummanipulation

- Abschneiden unerwünschter Suchpfade
- z.B. Erzwingen eines eindeutigen Berechnungsergebnisses
- spezielles nullstelliges Prädikat $! / 0$ (cut)
- Semantik:
 - legt die Suche auf diejenige Auswahl fest, die seit der Unifikation des Klauselkopfes entstanden ist, in der der cut auftritt.
 - ist immer erfolgreich.

Suchraummanipulation

- schneidet alle Suchpfade ab,
 - die sich aus alternativen Klauseln für das betreffende Prädikat ergeben, bzw.
 - die sich aufgrund alternativer Wertebindungen der bereits abgearbeiteten Teilziele im Klauselkörper ergeben.
- verhindert ein Backtracking in Teilziele, die links vom `cut` stehen
- macht alle Klauseln, die in der Datenbank auf den `cut` folgen unsichtbar

Suchraummanipulation

$a(X) :- b(X,Y), c(Y).$

$a(X) :- d(X).$

$b(a,b).$

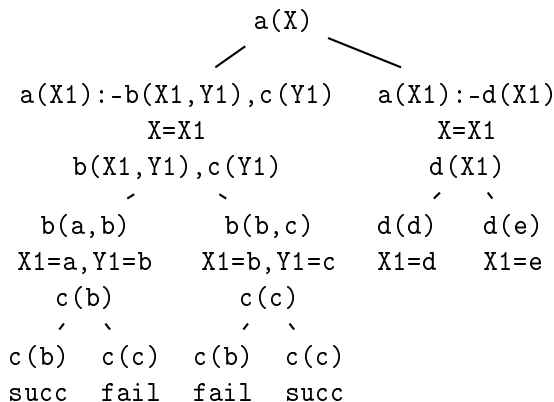
$b(b,c).$

$c(b).$

$c(c).$

$d(d).$

$d(e).$



Suchraummanipulation

`a(X) :- b(X,Y), !, c(Y).`

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

`b(a,b).`

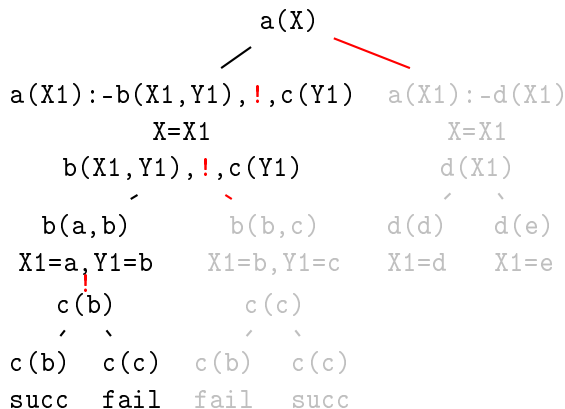
`b(b,c).`

`c(b).`

`c(c).`

`d(d).`

`d(e).`



Suchraummanipulation

Pragmatik:

- green cut: deterministische Programmierung in eindeutigen Kontexten
 - Effizienzsteigerung bei unveränderter Prädikatssemantik
- red cut: eine mehrdeutige Relation wird funktional
 - immer mit Änderung der Programmsemantik verbunden

Beispiele: green cut

- Verhindern unnötiger Klauselaufrufe bei disjunkten Fallunterscheidungen

```
% maximum (+X,+Y,?Max)
% Max ist Max von X und Y
maximum(X,Y,X) :- X>Y.
maximum(X,Y,Y) :- Y>=X.
```

- falls erste Klausel erfolgreich, scheitert zweite Klausel zwangsläufig

```
% maximum (+X,+Y,?Max)
% Max ist Max von X und Y
maximum(X,Y,X) :- X>Y, !.
maximum(X,Y,Y) :- Y>=X.
```

Beispiele: green cut

- kann Bedingung in zweiter Klausel ganz entfallen?

```
% maximum1 (+X,+Y,?Max)
% Max ist Max von X und Y
maximum1(X,Y,X) :- X>Y, !.
maximum1(X,Y,Y).
```

- Verlust einer Instanziierungsvariante!

```
?- maximum1(3,2,M).
```

```
    M = 3.
```

```
?- maximum1(3,2,3).
```

```
    true.
```

```
?- maximum1(3,2,2).
```

```
    true.
```

- → red cut
- korrigiertes Prädikatsschema:
maximum1(+X,+Y,-Max)
- vgl. if-then-else

Beispiele: green cut

- kann Bedingung in zweiter Klausel ganz entfallen?

```
% maximum1 (+X,+Y,?Max)  
% Max ist Max von X und Y  
maximum1(X,Y,X) :- X>Y, !.  
maximum1(X,Y,Y).
```

- Verlust einer Instanziierungsvariante!

```
?- maximum1(3,2,M).
```

```
    M = 3.
```

```
?- maximum1(3,2,3).
```

```
    true.
```

```
?- maximum1(3,2,2).
```

```
    true.
```

- → red cut
- korrigiertes Prädikatsschema:
 maximum1(+X,+Y,-Max)
- vgl. if-then-else

Beispiele: red cut

- Verhindern unnötiger Klauselaufrufe bei garantiert eindeutigem Ergebnis

```
% ist_mutter(+name)  Instanziierungsabhängig!  
% name ist Mutter  
ist_mutter(Mutter) :-  
    elternteil_von(Mutter,_),  
    weiblich(Mutter), !.
```

- cut verhindert aussichtslose Suche nach anderen Müttern
- aber: relationaler Charakter geht verloren
Aufzählen der Mütter ist nicht mehr möglich

Beispiele: red cut

- Verlust von alternativen Berechnungsergebnissen

```
% mutter_von(?name1,?name2)
mutter_von(Mutter,Kind) :-
    elternteil_von(Mutter,Kind),
    weiblich(Mutter), !.
```

- terminiert nach dem ersten Suchergebnis

```
elternteil_von(susi,hans).
elternteil_von(susi,anna).
```

```
?- mutter_von(susi,X).
    X = hans.
```

Anwendung: Defaultschließen

- Normalfall wird durch Ausnahmen außer Kraft gesetzt
- Normalfall: Tiere haben 6 Beine

```
hat_beine(vogel,2).           % Ausnahme
hat_beine(spinne,8).          % Ausnahme
hat_beine(lurch,4).           % Ausnahme
hat_beine(saeugetier,4).       % Ausnahme
hat_beine(_,6).               % Normalfall
```


Anwendung: Defaultschließen

- reine Faktensammlung ergibt unerwünschte Mehrfachresultate

```
?- hat_beine(spinne,Beine).  
    Beine = 8 ;  
    Beine = 6.
```

```
?- hat_beine(Tier,4).  
    Tier = Lurch ;  
    Tier = Saeugetier.
```

Anwendung: Defaultschließen

- Unterdrücken der Mehrfachresultate durch Cut:

```
hat_beine(vogel,2) :- !.           % Ausnahme
hat_beine(spinne,8) :- !.         % Ausnahme
hat_beine(lurch,4) :- !.          % Ausnahme
hat_beine(primate,2) :- !.        % Ausnahme
hat_beine(saeugetier,4) :- !.     % Ausnahme
hat_beine(_,6).                   % Normalfall
```

```
?- hat_beine(spinne,Beine).
    Beine = 8.
```

```
?- hat_beine(Tier,4).
    Tier = lurch.
```

- aber: funktional in beiden Argumenten
- eine Instanziierungsvariante geht verloren
- Reihenfolge der Klauseln ist signifikant

Defaultschließen

- Defaultschließen ist ein wichtiger Spezialfall des nichtmonotonen Schließens

Monotonie

Durch die Hinzunahme neuer Axiome wächst die Zahl der ableitbaren Theoreme monoton

$$A_1 \subset A_2 \wedge A_1 \models T_1 \wedge A_2 \models T_2 \rightarrow T_1 \subseteq T_2$$

- bei der Erweiterung einer Theorie gehen keine Schlussfolgerungen verloren

Nichtmonotonie

Durch die Hinzunahme neuer Axiome verringert sich die Zahl der ableitbaren Theoreme

- bei der Erweiterung einer Theorie können Schlussfolgerungen verloren gehen

Monotonie

```
hat_beine(_,6).
```

```
?- hat_beine(regenwurm,B).  
    B = 6.
```

```
hat_beine(regenwurm,0).  
hat_beine(_,6).
```

```
?- hat_beine(regenwurm,B).  
    B = 0 ;  
    B = 6.
```

Nichtmonotonie

```
hat_beine(_,6).
```

```
?- hat_beine(regenwurm,B).  
    B = 6.
```

```
hat_beine(regenwurm,0) :- !.  
hat_beine(_,6).
```

```
?- hat_beine(regenwurm,B).  
    B = 0.
```

Datenbankmanipulation

- Eintragen von Klauseln in die Datenbank

`consult(+File)`

Einlesen aus File

`assert(+Term)`

Eintragen einer Klausel ...

`assert(+Term, -Reference)`

`asserta, assertz`

... am Anfang / ... am Ende

- Entfernen von Klauseln aus der Datenbank

`abolish(+Functor, +Arity)`

`retract(+Head)`

`retractall(+Head)`

- Suche von Klauseln in der Datenbank

`clause(?Head, ?Body)`

`nth_clause(?Predicate, ?Index, ?Reference)`

Globale Variable

- globale Variable

```
% flag(+Key,-OldInteger,+NewInteger)}
```

```
?- flag(mycounter,N,N+1)
```

```
    N = 0.
```

```
?- flag(mycounter,N,N+1).
```

```
    N = 1.
```

```
?- flag(mycounter,N,N+1).
```

```
    N = 2.
```

- auch für nichtnumerische Atome

Prädikate höherer Ordnung

Meta-
level

```
pip(true) :- !.  
pip((G1,G2)) :- !,  
    pip(G1), pip(G2).  
pip(Goal) :-  
    clause(Goal,Body),  
    pip(Body).
```

```
mylast([E],E).
```

```
mylast([_:R],E) :-
```

```
    mylast(R,E).
```

Base-
level

8. Prädikate höherer Ordnung

Prädikatsaufruf

Steuerstrukturen

Resultatsaggregation

Metaprogrammierung

Prädikate höherer Ordnung

- ... erlauben Prädikatsaufrufe als Argumentbelegungen *und aktivieren diese als (Teil-)ziel*
 - Prädikatsaufruf
 - Negation
 - Steuerstrukturen
 - Ermittlung von Resultatsaggregationen
 - Konsistenzforderungen
 - ...
- ... können Prädikatsdefinitionen erzeugen und in die Datenbasis eintragen (über `assert/1`)
 - aber: Prädikatsdefinition erfolgt immer klauselweise!

Prädikatsaufruf

- `call(+Struktur)`
- Semantik: startet die Suche mit einem (möglicherweise underspezifizierten) Ziel (Struktur)
- Pragmatik: sinnvoll insbesondere:
 - im Zusammenhang mit dynamisch erzeugten Prädikaten und Prädikatsaufrufen
 - zur verzögerten Aktivierung von Zielen
→ Metaprogrammierung

Prädikatsaufruf

- Metacall: ist ein Teilziel im Körper einer Klausel eine Variable, so wird ihr Wert als Prädikatsaufruf interpretiert.

```
mycall(P) :- P.
```

```
?- mycall(length([a,b,c],X)).  
    X = 3.
```

- Beispiel: Definition der Disjunktion

```
% +X ; +Y:  
% Disjunktion von zwei Zielen X und Y  
X ; Y :- X.  
X ; Y :- Y.
```

Prädikatsaufruf

- `once(+Struktur)`
- Aufruf eines Prädikats ohne Backtracking

```
% once(+Ziel)  
% Ruft Ziel ohne Backtracking  
once(Goal) :- Goal, !.
```

- oftmals in eindeutige Versionen nichtdeterministischer Prädikate eingebaut, z.B.

```
memberchk(Elem,Liste) :-  
    once(member(Elem,Liste)).
```

Negation

- einstelliges Prädikat `\+/1` bzw. `not/1`
`\+/1` auch als Präfixoperator vordefiniert
- `\+ Ziel` ist wahr, wenn der Aufruf von `Ziel` scheitert

```
% \+(+Ziel)
% Prüft, ob Ziel scheitert
'\+'(Goal) :- Goal, !, fail.
'\+'(_).
```

```
?- \+(1=2).
```

```
true.
```

```
?- \+(1=1).
```

```
false.
```

Negation

- $\backslash + / 1$ instanziiert sein Argument nicht!
 - Frage nach den Bedingungen, die ein Ziel scheitern lassen, ist nicht möglich.
 - doppelte Negation ändert das Systemverhalten.

```
?- last(X, [a,b,c]).  
X = c.
```

```
?- \+ last(X, [a,b,c]).  
false.
```

```
?- \+ \+ last(X, [a,b,c]).  
X = _G281.
```

Negation

- Die Negation prüft nur das Scheitern eines Ziels (negation as failure).
- Voraussetzung: Annahme einer abgeschlossenen Welt (closed world assumption)
- Negation as failure ist der einfachste Fall einer nichtmonotonen Logik (vgl. auch `cut/0`)

Negation

- Monotonie: Hinzunahme von zusätzlichen Axiomen führt nicht zum Verlust von Theoremen:

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

```
b(a).
```

```
?- a(a).
```

```
true.
```

```
?- a(b).
```

```
false.
```

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

```
b(a).
```

```
b(b).
```

```
?- a(a).
```

```
true.
```

```
?- a(b).
```

```
true.
```


Negation

- Nichtmonotonie: Hinzunahme von Axiomen führt zum Verlust von Theoremen

<code>a(X):- \+ b(X).</code>	<code>a(X) :- \+ b(X).</code>
<code>b(a).</code>	<code>b(a).</code>
	<code>b(b).</code>
<code>?- a(a).</code>	<code>?- a(a).</code>
<code> false.</code>	<code> false.</code>
<code>?- a(b).</code>	<code>?- a(b).</code>
<code> true.</code>	<code> false.</code>

Steuerstrukturen

- Grundlegende Steuerstrukturen
 - Guarded Clauses
 - Rekursion
 - Cut
- zusätzliche Konstrukte
 - Konditionale
 - Iterationszyklen

Konditionale

- if-then

```
% +Condition -> +Action  
If -> Then :- If, !, Then.
```

Achtung: wenn If scheitert, scheitert das gesamte Konditional!

- if-then-else:

```
% +Condition -> +Then_Act ; Else_Act  
If -> Then ; _ :- If, !, Then.  
_ -> _ ; Else :- !, Else.
```

Konditionale

- Soft-Cut: $*\rightarrow$ If-Ziel bleibt backtrackbar

a(1). ?- a(X) \rightarrow Y=a ; Y=b.

a(2). X=1, Y=a.

?- a(4) \rightarrow Y=a ; Y=b.
Y=b.

?- a(X) $*\rightarrow$ Y=a ; Y=b.
X=1, Y=a ;
X=2, Y=a.

?- a(4) $*\rightarrow$ Y=a ; Y=b.
Y=b.

Failure-gesteuerte Zyklen

- über eine Folge natürlicher Zahlen (between/3)

```
% tab(+Integer)
tab(N) :-
    between(1,N,_),
    put(' '),
    fail.
```

- über eine Kollektion

```
% write_items(+List).
write_items(List) :-
    member(X,List),
    write_ln(X),
    fail.
```

Failure-gesteuerte Zyklen

- über der Datenbank

```
loop :-  
    mitarbeiter(Vorname,_,_,_,_),  
    write_ln(Vorname),  
    fail.  
  
?- loop.  
susi  
hans  
...  
karl  
false.
```

Failure-gesteuerte Zyklen

- Failure-gesteuerte Zyklen scheitern immer
 - ggf. muss alternative Klausel bereitgestellt werden

```
% write_items(+List).  
write_items(List) :-  
    member(X,List),  
    write_ln(X),  
    fail.  
write_items(_) :- true.
```

Iteration

- Iterator: forall(+Cond,+Goal)
- Iteration über eine Folge natürlicher Zahlen

```
% tab(+Integer)
tab(N) :-
    forall(between(1,N,_),
           put(' ')).
```

- Äquivalent zur for-Schleife
- Iteration über eine Kollektion

```
% write_items(+List).
write_items(List) :-
    forall(member(X,List),
           write_ln(X)).
```


Iteration

- Überprüfung von Konsistenzforderungen: Erfüllen alle Resultate des Aufrufs von Ziel, die Bedingung?

```
?- forall(mitarbeiter(Name,Gehalt),Gehalt>0).  
    true.
```

```
?- forall(member(E,[1,3,-2,1,2]),E>0).  
    false.
```

Iteration

- forall/2 instantiiert keine Variablen
- Implementation durch doppelte Negation

```
forall(Cond, Goal) :-  
    \+ (Cond, \+ Goal).
```

- Falls eine Variablenbindung "nach außen" gegeben werden soll:
foreach/2

Iteration

- `foreach(+Cond,+Goal)` ruft eine Konjunktion aller Instanzierungsvarianten von `Goal` auf, die sich aus dem Aufruf von `Cond` ergeben

```
?- foreach(between(1,3,X),member(X,Y)).  
   Y = [1, 2, 3|_G549] ;  
   Y = [1, 2, _G548, 3|_G552] ;  
   Y = [1, 2, _G548, _G551, 3|_G555] ;  
   Y = [1, 2, _G548, _G551, _G554, 3|_G558] .
```

- zum Vergleich

```
?- forall(between(1,3,X),member(X,Y)).  
   true.
```

Unbeschränkte Iteration

- Anwendung für Interaktionszyklen
Abbruch bei Eingabe von vereinbartem Endekennzeichen

```
loop :-  
    repeat,  
    read(X),  
    echo(X), !.  
  
echo(X) :- last_input(X), !.      % Abbruch  
echo (X) :- write_ln(X), fail.  
  
last_input(end).
```

Iteration

- Iteration erlaubt keinen Bezug auf vorangegangene Berechnungsergebnisse
- sinnvoll nur ...
 - ... falls Berechnungsergebnis irrelevant
 - Überprüfung von Konsistenzbedingungen
 - ... im Zusammenhang mit Nebeneffekten
 - Input/Output
 - Manipulation der Datenbasis
 - Verwendung globaler Variablen

Resultatsaggregation

- Aufsammeln aller Lösungen für ein Ziel:
`findall(+Term,+Ziel,-Liste)`
- Semantik: sammelt alle Instanziierungsvarianten, die beim Aufruf von `Ziel` sukzessive für den Term `Term` erzeugt werden, in der Liste `Liste`.
- Pragmatik: `Term` sollte Variable enthalten, die mit Variablen in `Ziel` koreferenzieren

- die Datenbank

```
% ma(Vorname,Name,Abteilung,Position,Gehalt)
ma(susi,sorglos,verwaltung,sekretaerin,40000).
ma(hans,im_glueck,verwaltung,manager,900000).
ma(anne,pingelig,rechenzentrum,operator,50000).
ma(paul,kraft,montage,wartung,70000).
ma(karl,wunderlich,versand,fahrer,55000).
```

Resultatsaggregation

```
?- findall(name(Vorname,Name),  
    ma(Vorname,Name,_,_,_),  
    Namen).  
Name = _G123,  
Vorname = _G234,  
Namen =  
    [name(susi,sorglos), name(hans,im_glueck),  
     name(anne,pingelig), name(paul,kraft),  
     name(karl,wunderlich)].
```


Resultatsaggregation

```
?- findall(Name,  
    ma(_,Name,verwaltung,_,_),Namen).  
Name = _G123, Namen = [sorglos,im_glueck].  
  
?- findall(Abteilung,  
    (ma(_,_,Abteilung,_,Gehalt),Gehalt>50000),  
    Abteilungen).  
Abteilung = _G123,  
Gehalt = _G234,  
Abteilungen = [verwaltung, montage, versand].
```

Resultatsaggregation

```
?- findall(Gehalt,  
    ma(_,_,_,_,Gehalt),Gehaelter),  
    min-list(Minimum,Gehaelter).  
Gehalt = _G123,  
Gehaelter = [40000,900000,50000,70000,55000],  
Minimum = 55000.
```

```
?- findall(Name,  
    ma(_,Name,verwaltung,_,_),  
    Namen), length(Anzahl,Namen).  
Name = _G123, Namen = [sorglos, im_glueck],  
Anzahl = 2.
```

Resultatsaggregation

- Aufsammeln aller Lösungen gruppiert nach identischen Bindungen für freie Variable: `bagof(+Term,+Ziel,-Liste)`

```
?- bagof(m(V,N,P,G),ma(V,N,A,P,G),L).  
   N = _G123, ... , A = verwaltung,  
       L = [m(susi, sorglos, sekretaerin, 40000),  
            m(hans, im_glueck, manager, 900000)] ;  
   N = _G123, A = rechenzentrum,  
       L = [m(anne, pingelig, operator, 50000)] ;  
   N = _G123, ... , A = montage,  
       L = [m(paul, kraft, wartung, 70000)] ;  
   N = _G123, ... , A = versand,  
       L = [m(karl, wunderbar, fahrer, 55000)].
```

Resultatsaggregation

- Ignorieren irrelevanter Variablenbelegungen:
 - neuer Infixoperator: $+Variable \hat{+} Ziel$

?- bagof(V, N \hat{P} G \hat{ma} (V,N,A,P,G), L).

N = _G123, ..., A = verwaltung,
L = [susi,hans] ;

N = _G123, ..., A = montage,
L = [paul] ;

N = _G123, ..., A = rechenzentrum,
L = [anne] ;

N = _G123, ..., A = versand,
L = [karl].

Resultatsaggregation

- Ermitteln von Lösungsmengen: `setof(+Term,+Ziel,-Liste)`
- Semantik: Wie `bagof/3`, aber Ergebnislisten sind lexikalisch sortiert und duplikatenfrei

```
?- setof(V, N^P^G^ma(V,N,A,P,G), L).  
    N = _G123, ..., A = verwaltung,  
      L = [hans,susi] ;  
    N = _G123, ..., A = montage, L = [paul] ;  
    N = _G123, ..., A = rechenzentrum,  
      L = [anne] ;  
    N = _G123, ..., A = versand, L = [karl].
```

Resultatsaggregation

- verallgemeinerte Resultatsaggregation:

`aggregate_all(+Template,+Goal,?Result)`

- Beispiele für Templates

<code>count</code>	zählt die Anzahl der Lösungen
<code>sum(Expr)</code>	ermittelt die Summe aller Lösungen für Expr
<code>min(Expr)</code>	ermittelt das Minimum aller Lösungen für Expr
<code>max(Expr)</code>	ermittelt das Maximum aller Lösungen für Expr
<code>bag(X)</code>	berechnet eine Liste aller Lösungen für X
<code>set(X)</code>	berechnet eine geordnete Menge aller Lösungen für X

Resultatsaggregation

```
?- aggregate_all(count,member(X,[1,2,3,2,1]),L).
```

```
L = 5.
```

```
?- aggregate_all(min(X),member(X,[1,2,3,2,1]),L).
```

```
L = 1.
```

```
?- aggregate_all(max(X),member(X,[1,2,3,2,1]),L).
```

```
L = 3.
```

```
?- aggregate_all(sum(X),member(X,[1,2,3,2,1]),L).
```

```
L = 9.
```

```
?- aggregate_all(set(X),member(X,[1,2,3,2,1]),L).
```

```
L = [1, 2, 3].
```

```
?- aggregate_all(bag(X),member(X,[1,2,3,2,1]),L).
```

```
L = [1, 2, 3, 2, 1].
```

Resultatsaggregation

- Variante `aggregate_all/4` erlaubt die Angabe eines Diskriminators an der zweiten Argumentposition, wodurch Mehrfachbindungen für eine Variable berücksichtigt werden

```
a(1,a).
```

```
a(2,a).
```

```
a(1,b).
```

```
a(2,b).
```

```
a(3,b).
```

```
?- aggregate_all(count,a(_,_),Anz).
```

```
    Anz = 5.
```

```
?- aggregate_all(count,X,a(X,_),Anz).
```

```
    Anz = 3.
```

```
?- aggregate_all(count,Y,a(_Y),Anz).
```

```
    Anz = 2.
```


Resultatsaggregation

- Verhindert, dass bei m:n-Relationen Ergebnisse verloren gehen

```
?- aggregate_all(sum(X), a(X,Y), R) .  
    R = 9.
```

```
?- aggregate_all(sum(X), X, a(X,Y), R) .  
    R = 6.
```

```
?- aggregate_all(sum(X), Y, a(X,Y), R) .  
    R = 9.
```

Metaprogrammierung

- Metaprogrammierung: Prolog-Programme verarbeiten andere Prolog-Programme als Daten
- Anwendungen:
 1. Analyse von Programmen
 - Pretty-Printer
 2. Transformation von Programmen
 - Programmoptimierung (z.B. Tailrekursion, delayed deduction)
 - Programmcompilierung
 - Übersetzen einer DCG → Differenzlistennotation
 3. Simulation der Programmabarbeitung: Meta-Interpreter

Metaprogrammierung

- Meta-Interpreter: Interpreter für eine Programmiersprache, der in der Programmiersprache selbst geschrieben ist
- einfachster Meta-Interpreter (`call/1`)
`rufe(Goal):-Goal.`

Prolog in Prolog

- Meta-Interpreter zur Simulation des Berechnungsmodells von Prolog
- Grundstruktur tritt in allen Anwendungen wieder auf
- eingeschränkte Syntax: Körper ist Konjunktion von positiven Literalen

Prolog in Prolog

```
% pip(+Ziel)
pip(true) :- !.
pip((G1,G2)) :-
    !,
    pip(G1),          % Zerlegung einer Konjunktion
    pip(G2).          % von Teilzielen
pip(Goal) :-          % Aktivierung eines Teilziels
    clause(Goal,Body),
    pip(Body).
```

```
?- pip(mylast(L,a)).
L = [a] ;
L = [X1, a] ;
L = [X2, X1, a] ;
...
```

Prolog in Prolog

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

Baselevel

Prolog in Prolog

Metalevel

```
pip(true) :- !.  
pip((G1,G2)) :- !,  
    pip(G1), pip(G2).  
pip(Goal) :-  
    clause(Goal,Body),  
    pip(Body).
```

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

Baselevel

Prolog in Prolog

Metalevel

```
pip(true) :- !.  
pip((G1,G2)) :- !,  
    pip(G1), pip(G2).  
pip(Goal) :-  
    clause(Goal, Body),  
    pip(Body).
```

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

Baselevel

Prolog in Prolog

Metalevel

```
pip(true) :- !.  
pip((G1,G2)) :- !,  
    pip(G1), pip(G2).  
pip(Goal) :-  
    clause(Goal, Body),  
    pip(Body).
```

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

Baselevel

Prolog in Prolog

- Anwendungen:
 - eigene Trace- und Debug-Werkzeuge
 - Interpreter mit veränderter operationaler bzw. denotationeller Semantik
 - modifizierte Inferenzregel
- Standardinferenzregel:
 - Ableitbarkeit eines Ziels
 - aus den in der Datenbasis abgespeicherten Klauseln und
 - mit Hilfe der SLD-Inferenzstrategie:
Top Down - Tiefe zuerst - Links-Rechts

Prolog in Prolog

- alternative Inferenzstrategien: Modifikation der operationalen Semantik
 - bottom-up Suche (z.B. Datalog, Parsing)
 - Breite-zuerst-Verarbeitung
 - parallele Bearbeitung von Teilzielen
- eingeschränkte Resolutionsverfahren: z.B. Zyklenüberwachung
- inferentiell erweiterte Klauselmengen: z.B. durch Vererbung (Modifikation der denotationellen Semantik)

Ablaufprotokollierung

- visible Prolog: Trace der Programmabarbeitung
 - Ausgabe der aktivierten Teilziele
 - Visualisierung der rekursiven Einbettung

Ablaufprotokollierung

```
% vip(+Ziel)
vip(G) :- vi('',G). % Init. der Einrücktiefe

vi(_,true) :- !.
vi(Indent,(G1,G2)) :-
    !,
    vi(Indent,G1),      % Zerlegung eines Teilziels
    vi(Indent,G2).
vi(Indent,G1) :-
    clause(G1,Body), % Suche einer relev. Klausel
    write(Indent),   % Ausgabe der Einrückung
    write('  ?- '),
    write(G1),nl,     % Ausg. des aktiven Teilziels
    atom_concat(Indent,' ', NewIndent), % Einrück.
    vi(NewIndent,Body). % rekursiver Aufruf
```

Ablaufprotokollierung

```
?- vip(mylast([a,b,c],E)).  
    ?- mylast([a, b, c],_G198)  
        ?- mylast([b, c]_G198)  
            ?- mylast([c],c)  
E = c ;  
    ?- mylast([c]_G198)  
false.
```

Ablaufprotokollierung

```
?- vip(mylast(X,a)).  
    ?- mylast([a], a)  
X = [a] ;  
    ?- mylast([_G255|_G256], a)  
        ?- mylast([a], a)  
X = [_G255, a] ;  
        ?- mylast([_G274|_G275], a)  
            ?- mylast([a], a)  
X = [_G255, _G274, a] ;  
            ?- mylast([_G293|_G294], a)  
                ?- mylast([a], a)  
X = [_G255, _G274, _G293, a] .
```

Ablaufprotokollierung

- Erweiterung:
 - Ausgabe der zum Vergleich herangezogenen Klauseln und des Unifikationsresultats
 - Protokollieren auch der erfolglosen Klauselaufrufe
- Trennen:
 - Suche nach einer relevanten Klausel
 - Unifikation mit dem Klauselkopf

Ablaufprotokollierung

```
vi2(Indent,G1) :-  
    % Erzeugen eines uninstantiierten Teilziels  
    G1=..[Pred|Args1], % functor(G1,F,N)  
    length(Args1,Length),  
    length(Args2,Length),  
    G2=..[Pred|Args2], % functor(G2,F,N)  
    % Suche nach einer passenden Klausel  
    find_clause(Indent,G2,Body,Pred),  
    write(Indent), write('  ?- '), write(G1),  
    % Ausgabe: Klauselkopf  
    write('    clause: '),  
    write(G2),  
    % Matching des Klauselkopfes  
    (G1 = G2 ->  
        write('    S'), nl ;  
        write('    F'), nl, fail),  
    concat(Indent,' ', NewIndent),  
    vi2(NewIndent,Body).
```

Ablaufprotokollierung

- Finden einer relevanten Klausel

```
% find_clause(+Indent,+Query,-Body,  
%      +PredicateNameOfQuery)  
find_clause(_,Q,Body,_) :-  
% Suche einer passenden Klausel  
    clause(Q,Body).  
find_clause(Indent,_,_,Pred) :-  
% Klauselmenge vollstaendig abgearbeitet  
    write(Indent),  
    write(': End of definition: '),  
    write(Pred), nl, fail.
```

Ablaufprotokollierung

```
?- vip2(mylast(E,[a,b,c])).  
  ?- mylast([a, b, c],_G204)    clause: mylast([_G321],_G321)    F  
  ?- mylast([a, b, c],_G204)    clause: mylast([_G345|_G346],_G321)    S  
    ?- mylast([b, c],_G204)    clause: mylast([_G361],_G361)    F  
    ?- mylast([b, c],_G204)    clause: mylast([_G385|_G386],_G361)    S  
      ?- mylast([c],_G204)    clause: mylast([_G401],_G401)    S  
E = c ;  
  ?- mylast([c],_G204)    clause: mylast([_G425|_G426],_G401)    S  
    ?- mylast([],_G204)    clause: mylast([_G441],_G441)    F  
    ?- mylast([],_G204)    clause: mylast([_G465|_G466],_G441)    F  
      : End of definition: mylast  
    : End of definition: mylast  
  : End of definition: mylast  
false.
```

- Memoization:
 - Verwalten einer Abarbeitungsgeschichte (History): Liste aller bisher aktivierten Teilziele
 - Überprüfen, ob aktuelles Teilziel bereits in der History registriert wurde

Zyklenüberwachung

```
% ldi(+Goal)
% Zyklenerkennender Interpreter
ldi(Goal) :- ld([ ],Goal). % leere History
ld(_,true) :- !.
    % true ist immer wahr
ld(History,(Subgoal1,Subgoal2)) :- !,
    % Zerlegung einer Konjunktion
    ld(History,Subgoal1),
    ld(History,Subgoal2).
ld(History,Subgoal) :-
    % Bearbeitung eines einfachen Teilziels
    not(unifiable(Subgoal,History,3)),
    % keine Unifizierbarkeit mit drei
    % aufeinanderfolgenden Elementen der History
    clause(Subgoal,Body), % relevante Klausel
    ld([Subgoal|History],Body). % rekursiver Aufruf
    % mit erweiterter History und Klauselkörper
```

Zyklenüberwachung

- Hilfsprädikat: Unifizierbarkeit mit einem Element der Ableitungsgeschichte

```
% unifiable(+Goal,+History,+N)
% Goal ist mit N aufeinanderfolgenden
% Elementen der History unifizierbar
unifiable(_,_,0) :- !.
    % Abbruch: maximale Anzahl erreicht
unifiable(Goal,[Goal|Hrest],N) :-
    % rekurs. Abstieg mit Kons. der History
    N1 is N - 1, !,      % und der max. Anzahl
    unifiable(Goal,Hrest,N1).
```

Zyklenüberwachung

- Datenbasis

```
isa(a,b).
```

```
isa(b,c).
```

```
isa(X,Z):-isa(X,Y),isa(Y,Z).
```

- Testaufrufe

```
?- isa(a,X).
```

```
    X = b ;
```

```
    X = c ;
```

```
    ...
```

```
?- ldi(isa(a,X)).
```

```
    X = b ;
```

```
    X = c.
```

Zyklenüberwachung

- Modifizierte Inferenzregel:
- Ableitbarkeit eines Ziels
 - aus den in der Datenbasis abgespeicherten Klauseln und
 - mit Hilfe der SLD-Inferenzstrategie, falls dabei nicht mehr als n-mal in direkter Folge jeweils miteinander unifizierbare Teilziele aktiviert werden
- Lösung deckt nicht alle Fälle ab
 - nur Zyklen, die durch direkte Rekursion entstehen
- Lösung ist zu rigide:
 - Zyklen zur Erzeugung von Nebeneffekten werden auch unterbunden

- Beispiel für indirekte Rekursion:

```
isa(a,b).
```

```
isa(b,c).
```

```
isa(X,Y):-is1(X,Y).
```

```
is1(X,Y):-isa(X,Z),isa(Z,Y).
```

```
?- ldi(isa(a,X)).
```

```
X = b ;
```

```
X = c ;
```

```
. . .
```

$$(\lambda(x).x + 2)$$
$$((\lambda(x).x^2)$$
$$3)$$
$$\Rightarrow 11$$

9. Funktionale Programmierung

Grundbegriffe

Umgebungen

Funktionale Auswertung

Rekursive Funktionen

Funktionen höherer Ordnung

Ein abschließender Vergleich

Funktionale Programmierung

- Funktion: eindeutige Abbildung aus einer Definitionsmenge in eine Zielmenge

$$y = f(x) \qquad x \mapsto y$$

- Applikation: Auswertung funktionaler Ausdrücke
 - Anwendung der Funktionsdefinition auf ein Wertetupel
 - Ermittlung des Funktionswerts für die gegebenen Argumentbelegungen
- Funktionale Programmierung: Berechnung durch Ermittlung von Funktionswerten

Scheme

- quasi-Standard: Revised Report on the Algorithmic Language Scheme (KELSEY ET AL. 1998)
- möglichst einfach gehalten
- viele Einzelheiten unspezifiziert → zahlreiche Implementationsunterschiede
- Systeme im Informatik-Rechenzentrum
 - MIT-Scheme
 - Dr. Scheme

Scheme

- (leicht) vereinfachte Syntax

Ausdruck	:=	Konstante Name s-Ausdruck
Konstante	:=	Zahl Wahrheitswert
Wahrheitswert	:=	#t #f
Variable	:=	Name
s-Ausdruck	:=	Liste gepunktetes Paar
Liste	:=	'()' '(' Elemente ')'
Elemente	:=	Element Elemente
Element	:=	Ausdruck
gepunktetes Paar	:=	'(Element '.' Element)'

- keine Makros
- keine reservierten Symbole für `quote` (und `backquote`)

Scheme

- Wertesemantik
- Konstanten haben sich selbst als Wert

```
2 ==> 2
```

```
#t ==> #t
```

- Variablen haben den Wert, der Ihnen (z.B. mit `define`) zugewiesen wurde

```
x ==> error: unbound variable
```

```
(define x 2) ==> #undefined
```

```
x ==> 2
```

- der Wert von s-Ausdrücken muss *berechnet* werden (z.B. durch funktionale Applikation)

Scheme in Prolog (1): Auswertung

- zentrales Auswertungsprädikat `eval/3`
- wird sukzessive definiert

`eval/3` (1): Auswertung einer Konstanten

```
%%% eval(Expression, Environment, Value)
% self-evaluating expressions
eval(Exp,_,Exp) :- number(Exp), !.
eval('#t',_, '#t') :- !.
eval('#f',_, '#f') :- !.
```

```
?- eval(2,_,X).
```

```
    X = 2.
```

```
?- eval('#t',_,X).
```

```
    X = '#t'.
```

Scheme in Prolog (2): Notation

- Notation
 - Lisp-Notation ist historischer Unfall
 - Leerzeichen ist überladen:
 - Separation von Listenelementen (\rightarrow Komma)
 - Füllmaterial zwischen Syntaxelementen (\rightarrow Leerzeichen)

\rightarrow Repräsentation von s-Ausdrücken als Prolog-Listen

Scheme	2	name	Name	(a b c)	(a . b)
--------	---	------	------	---------	---------

P-Scheme	2	name	name	[a,b,c]	[a b]
----------	---	------	------	---------	-------

Variable

- Ein Name identifiziert eine Variable (Symbol), deren Wert ein (abstraktes) Objekt ist.
- Benennung als einfachste Form der Abstraktion: Man sieht einem Wert nicht mehr an, wie er entstanden ist.

```
(define <name> <konstante>)
```

```
(define <name> <s-expression>)
```

- `define` ist eine *special form expression*
- Wert des `define`-Aufrufs ist undefiniert

Variable

- die funktionale Programmierung unterstützt im Prinzip keine Wertmanipulation ...
- ... lässt aber verschiedene Hintertüren offen: `define`, `set!`
- beide entsprechen der Wertzuweisung in den imperativen Sprachen
- Namen dürfen nicht doppelt belegt werden
- betrifft insbesondere
 - Standardfunktionen: `sqrt`, `sin`, `append`, ...
 - Syntaktische Schlüsselwörter: `and`, `begin`, `case`, `cond`, `define`, `delay`, `do`, `else`, `if`, `lambda`, `let`, `let*`, `letrec`, `or`, `quasiquote`, `quote`, `set!`, ...
 - Konstanten: `#t` (true), `#f` (false)
- Interpunktionszeichen: `'(, ')`, `'''` und `''` können nicht Teil eines Namens sein

s-Ausdrücke

Funktionsaufrufe (in Präfix-Notation)

`<(<Funktionsname> . <Liste von Argumenten>)>`

`(sqrt (+ (expt x 2) (expt y 2)))`

`[sqrt, [+ , [expt, x, 2], [expt, y, 2]]]`

lambda-ausdrücke

`(lambda <Liste von Argumenten> . <Liste von s-Ausdrücken>)`

`(lambda (x y) (sqrt (+ (expt x 2) (expt y 2))))`

`[lambda, [x, y], [sqrt, [+ , [expt, x, 2], [expt, y, 2]]]]`

special form expressions

special form expressions definieren eine spezielle Auswertungsreihenfolge

- `define` ist eine *special form expression*

Funktionen

- Funktionsdefinition als Abstraktion
- Funktionen müssen keinen Namen haben
Name wird "nur" für Dokumentation, Wiederverwendung und den rekursiven Aufruf benötigt
- lambda-Ausdrücke sind anonyme (unbenannte) Funktionen

Funktionen

Verwendung von lambda-Ausdrücken zur Definition einer benannten Funktion

```
(define pythagoras (lambda (x y)
  (sqrt (+ (expt x 2) (expt y 2)))))
```

```
[define, pythagoras, [lambda, [x, y],
  [sqrt, [+ , [expt, x, 2], [expt, y, 2]]]]]
```

Kurzversion ohne lambda

```
(define (pythagoras x y)
  (sqrt (+ (expt x 2) (expt y 2))))
```

```
[define, [pythagoras, x, y],
  [sqrt, [+ , [expt, x, 2], [expt, y, 2]]]]
```

Zuweisung eines lambda-Ausdrucks zu einem Namen

Funktionsauswertung

eval: Auswerten *aller* Listenelemente

apply: Anwenden des Evaluationsergebnisses für das erste Listenelement auf die Evaluationsergebnisse der Restliste

- Funktionsdefinition:

```
(define (square x) (* x x))
```

- Funktionsanwendung:

```
( square      (+ 3 1 2) )  
  (lambda (x)      6  
    (* x x) )  
                    36
```

Umgebungen

- Umgebungen sind Gültigkeitsbereiche für Variable
- jeder Funktionsaufruf eröffnet eine neue Umgebung und bindet die aktuellen an die formalen Parameter
- Funktionsaufrufe sind hierarchisch eingebettet → Umgebungen sind rekursiv verschachtelt
- Sichtbarkeit von Variablen: bottom-up Suche nach einer Variablenbindung

Umgebungen

- lokale Umgebungen können zusätzliche Variable einführen, die nicht formale Parameter sind
- $(\text{let } (\langle \text{Name-Wert-Paar}_1 \rangle \dots \langle \text{Name-Wert-Paar}_n \rangle) \langle \text{Funktionsaufruf} \rangle)$
- falls ein Ausdruck in Name-Wert-Paar_i auf einen Namen in einem Name-Wert-Paar_j mit $j < i$ Bezug nimmt, muss statt `let` `let*` verwendet werden.

Verschattung

```
(define x 3)
(define y 4)

(define (pythagoras x y)
  (let*
    ((square (lambda (z) (* z z)))
     (x2 (square x))
     (y2 (square y)) )
    (sqrt (+ x2 y2)) ) )

(pythagoras (+ x 3) (+ y 2))
```

Verschattung

```
[define, x, 3]
[define, y, 4]

[define, [pythagoras, x, y],
  [let_star,
    [[square, [lambda, [z], [* , z, z]]]
     [x2, [square, x]],
     [y2, [square, y]] ],
    [sqrt, [+ , x2, y2]] ] ]

[pythagoras, [+ , x, 3], [+ , y, 2]]
```

Umgebungen

```
(define x 3)
```

```
(define y 4)
```

```
(define (pythagoras x y)
  (let*
    ((square (lambda (z)
                 (* z z)))
     (x2 (square x))
     (y2 (square y)))
    (sqrt (+ x2 y2))))
```

```
(pythagoras
 (+ x 3)
 (+ y 2))
```

Globale Umgebung:

Standard-Namen, pythagoras, x, y

Umgebung von pythagoras:

x, y

Umgebung von let*:

square, x2, y2

Scheme in Prolog (3): Umgebungen

- Umgebungen implementiert als Fakten in der Datenbasis
- Variablenbindungen

```
% env(Environment, Variable, Value).  
env(e11,x,6).  
env(e11,y,6).  
env(e11,square,[lambda,[x],[ '*', x, x]]).  
...
```

- rekursive Einbettung von Umgebungen

```
% env(Sub_environment,Super_environment).  
env(e12,e11).  
env(e11,global).  
...
```

- `global` wird als oberste Umgebung vereinbart

Scheme in Prolog (3): Umgebungen

- die Prädikate `env/2` und `env/3` müssen dynamisch veränderbar und initial leer sein

```
:- dynamic(env/2).  
:- dynamic(env/3).  
:- retractall(env(_, _)).  
:- retractall(env(_, _, _)).
```

Scheme in Prolog (3): Umgebungen

Abfragen eines Variablenwerts

```
%%% get_value(Var,Env,Val)
% retrieve a variable value from the hierarchy of
% environments
get_value(Var,Env,Val) :-
    (env(Env,Var,Val) ->          % retrieve the value
     true ;                      % from the given environment
     ((Env\=global,env(Env,SEnv)) -> % look up the variable
      get_value(Var,SEnv,Val) ;    % one environment
      (write('no value for variable '), % higher up
       write(Var),
       write(' in environment '),
       writeln(Env)))).
```

Scheme in Prolog (3): Umgebungen

Eröffnen einer neuen Umgebung als Unterumgebung zu einer existierenden Umgebung

```
%%% extend_env(Variables,Values,Environment,NewEnvironment)
% extend the given environment with a new subenvironment
extend_env(Vars,Vals,Env,NewEnv) :-
    gensym(e,NewEnv),                % generate a name
    assert(env(NewEnv,Env)),         % insert in hierarchy
    bind_variables(Vars,Vals,NewEnv).
```

Hinzufügen von Variablenbindungen zu einer Umgebung

```
%%% bind_variables(Variables,Values,Environment)
% insert variable bindings into a new environment
bind_variables([],[],_).
bind_variables([Var|Vars],[Val|Vals],Env) :-
    assert(env(Env,Var,Val)),
    bind_variables(Vars,Vals,Env).
```


Scheme in Prolog (3): Umgebungen

Definieren eines Variablenwerts (define)

```
%%% define_variable(Variable,Value,Environment)
% define a new variable in a given environment
define_variable(Var,Val,Env) :-
    (env(Env,Var,_) -> retract(env(Env,Var,_)) ; true),
    assert(env(Env,Var,Val)).
```

Ändern eines Variablenwerts (set!)

```
%%% set_variable(Variable,Value,Environment)
% change the value of an already defined variable
set_variable(Var,Val,Env) :-
    env(Env,Var,_) ->
        (retract(env(Env,Var,_)), assert(env(Env,Var,Val))) ;
        (write('unknown variable '),writeln(Var)).
```

Scheme in Prolog (4): Auswertung

eval/3 (2): Auswertung einer Variablen

```
% variables
eval(Exp,Env,Val) :-
    atom(Exp), !,
    get_value(Exp,Env,Val).
```

```
?- define_variable(x,5,global).
    true.
?- eval(x,global,X).
    X = 5.
```

Funktionale Applikation

- Funktionsaufruf: s-Ausdruck

(Funktork, Argument₁, ..., Argument_n)

- funktionale Applikation

1. eval: Auswertung der Listenelemente

- erstes Element
→ Lambda-Ausdruck
- Auswertung der restlichen Elemente
→ aktuelle Parameter

2. apply: Anwendung des Lambda-Ausdrucks auf die Werte

Special form expressions

- *special form expressions*:
Abweichung von der Standard-Auswertungsreihenfolge
- müssen in jedem Einzelfall gesondert implementiert werden
- Grundsatz: der Funktor von *special form expressions* wird nie ausgewertet

Special form expressions

- Fall 1: Argumente werden nicht ausgewertet

`(quote <expression>)`

- Fall 2: Argumente werden nur teilweise ausgewertet

`(set! <variable> <expression>)`

`(define <name> <expression>)`

nur `<expression>` wird ausgewertet

Special form expressions

- Fall 3: Argumente werden bedingt ausgewertet

`(if <condition> <then> <else>)`

`<condition>` wird immer ausgewertet und in Abhängigkeit vom Resultat entweder der `<then>`- oder der `<else>`-Zweig

`(and <expr1> <expr2> ...)`

`(or <expr1> <expr2> ...)`

Auswertung der Argumente wird abgebrochen, sobald der Funktionswert ermittelt wurde

- Fall 4: Ausdruck wird transformiert

`cond` wird in geschachtelten `if`-Ausdruck umgewandelt

Scheme in Prolog (5): special form expressions

Fall 1: Argumente werden nicht ausgewertet

```
% quoted expressions  
eval([quote,Val],_,Val) :- !.
```

```
?- eval([quote, x],global,X).  
    X = x.
```

Scheme in Prolog (5): special form expressions

Fall 2: Argumente werden teilweise ausgewertet

zweites Listenelement ist ein Atom

```
% normal definitions
```

```
eval([define,Var,Valexpr],Env,ok) :- atom(Var), !,  
    eval(Valexpr,Env,Val),define_variable(Var,Val,Env).
```

zweites Listenelement ist eine Liste

```
% shorthand for procedure definitions
```

```
eval([define,[Var|Parms]|Valexpr],Env,ok) :- !,  
    define_variable(Var,[closure,Parms,Valexpr,Env],Env).
```


Scheme in Prolog (5): special form expressions

```
?- eval([define, x, 5],global,X).  
   X = ok.  
?- eval(x,global,X).  
   X = 5.  
?- eval([define, y, [+ , 1, 2]],global,X).  
   X = ok.  
?- eval(y,global,X).  
   X = 3.  
?- eval([define, [square, x], [* , x, x]],global,X).  
   X = ok.  
?- eval(square,global,X).  
   X = [closure, [x], [* , x, x], global].
```

Scheme in Prolog (5): special form expressions

Fall 2: Argumente werden teilweise ausgewertet (Forts.)

```
% assignments  
eval([setbang,Var,Valexpr],Env,ok) :- !,  
    eval(Valexpr,Env,Val),set_variable(Var,Val,Env).
```

```
?- eval([define, x, 5],global,X).  
   X = ok.  
?- eval([setbang, x, 6],global,X).  
   X = ok.  
?- eval(x,global,X).  
   X = 6.
```

Scheme in Prolog (5): special form expressions

Fall 3: Argumente werden bedingt ausgewertet

```
% conditionals without alternative
eval([if,Pred,Cons],Env,Val) :- !,
    eval(Pred,Env,Boolean),
    (true(Boolean) ->          % evaluiert Pred zu TRUE?
        eval(Cons,Env,Val)).

true('#t').

% conditionals with alternative
eval([if,Pred,Cons,Alter],Env,Val) :- !,
    eval(Pred,Env,Boolean),
    (true(Boolean) ->
        eval(Cons,Env,Val) ;
        eval(Alter,Env,Val)).
```

Scheme in Prolog (5): special form expressions

```
?- eval([define, x, 5],global,X).
```

```
    X = ok.
```

```
?- eval([if, [eq, x, 5], [quote, yes], [quote, no]],global,X).
```

```
    X = yes.
```

Scheme in Prolog (6): primitive Funktionen

Auswertung primitiver Funktionen

```
% function application for primitives
eval([Optor|Opnds],Env,Val) :-
    eval(Optor,Env,Optorval),
    primitive_proc(Optorval),!,
    list_of_values(Opnds,Env,Opndsval),
    apply_primitive(Optorval,Opndsval,Val),
    (flag(verbose,on,on) ->
        (write('into value '),writeln(Val)) ; true).
```

- Der Operator muss ausgewertet werden,
 - um auf eine Funktionsdefinition für ein Atom zuzugreifen, oder
 - um eine Funktionsdefinition dynamisch berechnen zu können.

Scheme in Prolog (6): primitive Funktionen

Auswertung primitiver Funktionen

```
%%% apply_primitive(Procedure,Arguments,Value)
% connects a primitive to an underlying Prolog-predicate
apply_primitive(+,Args,Value) :- sumlist(Args,Value).
apply_primitive(*,Args,Value) :- multlist(Args,Value).
apply_primitive(eq,[X,Y],Value) :-
    X==Y -> Value='#t' ; Value='#f'.
apply_primitive(car,[[Value|_]],Value).
apply_primitive(cdr,[_|Value]],Value).
apply_primitive(cons,[X,Y],[X|Y]).
apply_primitive(pair,[X],Value) :-
    X=[_|_] -> Value='#t' ; Value='#f'.
apply_primitive(null,[X],Value) :-
    X=[] -> Value='#t' ; Value='#f'.
apply_primitive(atom,[X],Value) :-
    atomic(X) -> Value='#t' ; Value='#f'.
```

Closures

- Funktionen sind "Bürger erster Klasse"
- Sie können als Wert einer Variablen zugewiesen werden ...

```
(define pythagoras (lambda (x y)
  (sqrt (+ (expt x 2) (expt y 2)))))
```

```
[define, pythagoras, [lambda, [x, y],
  [sqrt, [+ , [expt, x, 2], [expt, y, 2]]]]]
```

- ... als Wert an einer Argumentstelle übergeben werden ...

```
(apply pythagoras (3 4))
```

- ... oder als Resultat einer Berechnung auftreten

```
((list (quote lambda) (quote (x y))
  (quote (sqrt ...)) )
 3 4)
```

Closures

- Funktionsdefinitionen können freie und gebundene Variable enthalten
- Problemfall: freie Variable

Wertebindung zum Zeitpunkt der Funktionsdefinition muss nicht der Wertebelegung zum Zeitpunkt der Funktionsauswertung sein

- Lösung: Closure
 - jede Funktionsdefinition geschieht in einer Umgebung
 - das Paar aus Funktionsdefinition und seiner Definitionsumgebung heisst Closure

Scheme in Prolog (7): Closures

Erzeugen einer Closure

```
% lambda expressions
eval([lambda,Parms|Body],Env,[closure,Parms,Body,Env]) :- !.

% shorthand for procedure definitions (repeated here)
eval([define,[Var|Parms]|Valexpr],Env,ok) :- !,
    define_variable(Var,[closure,Parms,Valexpr,Env],Env).
```

Scheme in Prolog (8): Nutzerdefinierte Funktionen

Auswerten einer Funktion bzw. einer Closure

```
% procedures
eval([Opor|Opnds],Env,Val) :-
    eval(Opor,Env,[closure,Pparms,Body,Penv]),
        % retrieve the definition
    list_of_values(Opnds,Env,Opndsval),
        % evaluate the arguments
    extend_env(Pparms,Opndsval,Penv,Newenv),
        % create a new sub-environment
    eval_sequence(Body,Newenv,Val),
        % evaluate the procedure in the new environment
    retractall(env(Newenv,_,_)),
    retractall(env(Newenv,_)).
        % housekeeping: remove the used environment
```

Scheme in Prolog (8): Nutzerdefinierte Funktionen

Auswerten einer Funktion (Hilfsprädikate)

```
%%% list_of_values(expressions,environment,list_of_values)
% evaluate a list of expressions into a list of values
list_of_values([],_,[]).
list_of_values([Exp|Expns],Env,[Val|Vals]) :-
    eval(Exp,Env,Val),list_of_values(Expns,Env,Vals).

%%% eval_sequence(Expressions,Environment,Value)
% evaluate a sequence of expressions into a single value
eval_sequence([Exp],Env,Val) :- !, eval(Exp,Env,Val).
eval_sequence([Exp|Expns],Env,Val) :-
    eval(Exp,Env,_),eval_sequence(Expns,Env,Val).
```

Typetest Liste

```
(define (proper-list? liste)
  (if (equal? liste '()) #t
      (if (pair? liste) (proper-list? (cdr liste)) #f) ) )

proper_list([]).
proper_list([_|Rest]) :- proper_list(Rest).
```

Element einer Liste

```
(define (member elem liste)
  (if (null? liste) #f
      (if (equal? elem (car liste)) #t
          (member elem (cdr liste))) ) )
```

```
(define (member elem liste)
  (if (null? liste) #f
      (if (equal? elem (car liste)) (cdr liste)
          (member elem (cdr liste))) ) )
```

```
member(Elem, [Elem|_]).
```

```
member(Elem, [_|Rest]) :- member(Elem, Rest).
```

Rekursive Funktionen

Letztes Element einer Liste

```
(define (last liste)
  (if (null? (cdr liste)) (car liste) (last (cdr liste)) ) )

last(Elem,[Elem]).
last(Elem,[_|Rest]) :- last(Elem,Rest).
```

Länge einer Liste

```
(define (length liste)
  (if (null? liste) 0 (+ 1 (length (cdr liste))) ) )

length([],0).
length([_|Rest],Laenge) :-
  length(Rest,RLaenge),
  Laenge is RLaenge + 1.
```

Verketten zweier Listen

```
(define (append liste1 liste2)
  (if (null? liste1)
      liste2
      (cons (car liste1)
            (append (cdr liste1)
                    liste2)) ) )
```

```
append([],Liste,Liste).
append([Elem|Rest],Liste,[Elem|NeueListe]) :-
  append(Rest,Liste,NeueListe).
```

Umdrehen einer Liste

```
(define (reverse liste)
  (define (rev liste acc)
    (if (null? liste)
        acc
        (rev (cdr liste)
              (cons (car liste) acc)) ) )
  (rev liste '()) )
```

`reverse(Liste,RListe) :- rev(Liste,[],RListe).`

`rev([],RListe,RListe).`

`rev([Elem|Rest],Acc,RListe) :- rev(Rest,[Elem|Acc],RListe).`

Funktionen höherer Ordnung

- Funktionen, die Funktionen als Argumente fordern:
map, filter, reduce, ...
- Funktionen, die Funktionen als Argumente nehmen *und* Funktionen als Funktionswert zurückgeben:
curry, rcurry, compose, conjoin, ...

Funktionen höherer Ordnung

Transformieren aller Elemente einer Liste: map

```
(define (map* function liste)
  (if (null? liste)
      '()
      (cons (function (car liste) )
            (map* function (cdr liste))) ) )
```

```
(map* (lambda (x) (* 2 x) ) '(1 2 3) ) ==> (2 4 6)
```

```
(define (zero? x) (if (eq? x 0) #t #f) )
(map* zero? '(1 0 -1) ) ==> (#f #t #f)
```

Funktionen höherer Ordnung

Selektieren von Elementen einer Liste: filter

```
(define (filter* test liste)
  (if (null? liste)
      '()
      (if (test (car liste))
          (cons (car liste) (filter* test (cdr liste) ) )
          (filter* test (cdr liste) ) ) ) )
```

```
(filter* zero? '(1 0 -1)) ==> (0)
```

Funktionen höherer Ordnung

Erweitertes map mit beliebig vielen Argumenten

```
(define (map** function . liste-von-listen)
  (if (null? (filter* null? liste-von-listen))
      (cons (apply function (map* car liste-von-listen))
            (apply map**
                    function
                    (map* cdr liste-von-listen) ) )
      '( ) ) )
```

(map + '(1 2 3) '(2 3 4) '(3 4 5)) ==> (6 9 12)

Funktionen höherer Ordnung

Falten: reduce

Listenelemente paarweise mit einem Operator verknüpfen und auf einen Resultatswert reduzieren

```
(define (reduce function liste anfangswert)
  (if (null? liste)
      anfangswert
      (reduce function
                (cdr liste)
                (function anfangswert (car liste))) ) )
```

```
(reduce + (1 2 3 4 5) 0) ==> 15
```

```
(reduce + (map (lambda (x) 1) '(a b c)) 0) ==> 3
```

Funktionen höherer Ordnung

Funktionskomposition:

linkes Argument eines partiellen Funktionsaufrufs: `curry`

```
(define (curry function argument)
  (lambda (neues-argument)
    (function argument neues-argument) ) )
```

Funktionskomposition:

rechtes Argument eines partiellen Funktionsaufrufs: `rcurry`

```
(define (rcurry function argument)
  (lambda (neues-argument)
    (function neues-argument argument) ) )
```

```
((curry < 0) 1) ==> #t
((rcurry < 0) 1) ==> #f
```

Funktionen höherer Ordnung

- typische Verwendungsweisen von `curry`/`rcurry`

<code>(curry + 1)</code>	$f(x) = x + 1$	Inkrement
<code>(curry / 1)</code>	$f(x) = \frac{1}{x}$	Reziprokwert
<code>(rcurry / 2)</code>	$f(x) = \frac{x}{2}$	Halbieren
<code>(curry * 2)</code>	$f(x) = 2x$	Verdoppeln
<code>(rcurry expt 2)</code>	$f(x) = x^2$	Quadrieren
<code>(curry - 0)</code>	$f(x) = -x$	Vorzeichenumkehr
<code>(curry = 0)</code>	$f(x) = \begin{cases} \#t & \text{für } x = 0 \\ \#f & \text{sonst} \end{cases}$	Test gleich Null

- typischer Verwendungskontext

```
(map (rcurry > 0) '(1 0 -1)) ==> (#t #f #f)
```

- auch Erweiterung auf beliebig viele Argumente

Funktionen höherer Ordnung

Sukzessive Funktionsanwendung: `compose`

```
(define (compose func1 func2)
  (lambda (x) (func1 (func2 x))))
```

```
((compose car cdr) '(a b c)) ==> b
```

- weitere Funktionen

- `conjoin`: wahr wenn alle Prädikate wahr
- `disjoin`: wahr wenn mindestens ein Prädikat wahr
- `always`: erzeugt Funktion mit konstantem Wert

```
(reduce + (map (always 1) '(a b c)) 0) ==> 3
```


Funktionen höherer Ordnung

- Beispielanwendungen für Funktionen höherer Ordnung

```
(define (mean werte)
  (/ (reduce + werte 0)
     (reduce + (map (lambda (x) 1) werte)) ) )
```

```
(define (variance werte)
  (let (m (mean werte))
    (mean (map square (map (rcurry - m) werte)) ) ) )
```

```
(define (variance werte)
  (let (m (mean werte))
    (mean ((compose (curry map square)
                     (curry map (rcurry - m)))
           werte)) ) )
```

Logik-basiert und funktional: Ein Vergleich

logik-basiert und funktional

- leistungsfähige Abstraktionskonzepte
- flexible Sprachen zum funktionalen Prototyping
- schwache Typisierung, dynamische Datenstrukturen
- Interpretative Abarbeitung
- Erweiterbarkeit
- Metaprogrammierung

Logik-basiert und funktional: Ein Vergleich

logik-basiert

- leistungsfähige build-in-Konzepte (relationale DB, Unifikation, Suche)
- Richtungsunabhängigkeit
- extrem einfacher Variablenskopus
- intuitiv plausible Listenverarbeitung
- flexible Syntax

funktional

- systematische Softwarekomposition mit Funktionen höherer Ordnung
- gute Passfähigkeit zur objektorientierten Programmierung
- vollständige Integration der Arithmetik in das Verarbeitungsmodell

Logik-basiert und funktional: Ein Vergleich

- verschiedene Versuche zur Synthese
- z.B. Mercury (Universität Melbourne)
- Deklaration von Verarbeitungsmodi
- Compilation in ausführbaren Code für die verschiedenen Aufrufvarianten
- Reihenfolge der Abarbeitung von Teilzielen prinzipiell beliebig
- sequentielle Abhängigkeiten zwischen Teilzielen werden separat verwaltet (analog zur Differenzlistentechnik)

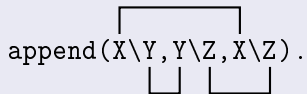


Diagram illustrating the append function: `append(X\Y, Y\Z, X\Z) .`

The diagram shows the list `X\Y` with a bracket underneath it, and the list `Y\Z` with a bracket underneath it. A larger bracket is positioned above both `X\Y` and `Y\Z`, spanning the width of both expressions, indicating that they are combined to form the result `X\Z`.

10. Aktive Datenstrukturen

Offene Listen

Differenzlisten

Definite Clause Grammar

dynamische Datenstrukturen

können sich bei Bedarf an unterschiedliche quantitative Anforderungen anpassen

- Listen, im Gegensatz zu Arrays

→ SE I

aktive Datenstrukturen

enthalten noch variable (instanziierbare) Elemente

- unbekannte Elemente, im Gegensatz zur Änderung von Elementen
- → unvollständige Datenstrukturen
- → offene Datenstrukturen

Abgeschlossene Listen

- Listenzugriff erfolgt immer vom Kopf aus
- → Modifikationen am Listenende erfordern vollständiges Kopieren der Liste
- Beispiel Stack: Zugriff nur vom Kopf aus

```
% stack(?Element,?AlterStack,?NeuerStack)
```

```
stack(E,Stack,[E|Stack]).
```

Abgeschlossene Listen

- drei Stackoperationen als unterschiedliche Instanziierungsvarianten

- push:

```
?- stack(a, [ ], S).
```

```
S = [a].
```

```
?- stack(b, [a], S).
```

```
S = [b, a].
```


Abgeschlossene Listen

- Stackoperationen (2)

- pop:

```
?- stack(E,S,[b,a]).  
    E = b, S = [a].
```

```
?- stack(E,S,[a]).  
    E = a, S = [ ].
```

- non-empty:

```
?- stack(_,[a]).  
    true.
```

```
?- stack(_,[ ]).  
    false.
```

Abgeschlossene Listen

- Beispiel Warteschlange: Zugriff auch am Listenende
 - **enqueue**: Einfügen am Listenende
 - **dequeue**: Wegstreichen am Listenanfang
- Einfügen und Wegstreichen sind keine Umkehroperationen mehr
→ nicht mehr in einem Prädikat realisierbar

Abgeschlossene Listen

- **enqueue**

```
% enqueue(?Element,?AlteQueue,?NeueQueue)
enqueue(E,[ ],[E]).
enqueue(E,[H|AR],[H|NR]) :- enqueue(E,AR,NR).
```

```
?- enqueue(a,[ ],Q).
   Q = [a].
```

```
?- enqueue(b,[a],Q).
   Q = [a,b].
```

Abgeschlossene Listen

- **dequeue**

```
% dequeue(?Element,?AlteQueuee,?NeueQueuee)  
dequeue(E, [E|R], R) .
```

```
?- dequeue(E, [a,b], Q) .  
    E = a, Q = [b] .
```

```
?- dequeue(E, [a], Q) .  
    E = a, Q = [ ] .
```

- Problem: Kopieren beim Einfügen

- Uninstanciierter Aufruf eines Listenarguments ist automatisch Listenkonstruktor
- Konsumtion von Freispeicherzellen

Offene Listen

- alternative Implementation als offene Liste
 - Listenrest ist eine uninstanciierte Variable
→ erlaubt das Hinzuunifizieren einer neuen Restliste
- Listenendedetektion durch Test auf leere Restliste:
 - leere Liste benötigt Sonderbehandlung
 - Listenelemente dürfen keine uninstanciierten Variablen mehr sein
- Repräsentation der Warteschlange

[]	[_ _]
[a]	[a _]
[a, b]	[a, b _]
[a, b, c]	[a, b, c _]
...	...

Offene Listen

- enqueue:

```
% enqu_ol(+Element,+Queue)
enqu_ol(E,[Y|X]) :-          % Einfuegen in die
    var(Y),var(X),Y=E.      % leere Liste
enqu_ol(E,[Y|X]) :-          % Einfuegen einer
    nonvar(Y),var(X),X=[E|_]. % neuen Restliste
enqu_ol(E,[_|T]) :-          % Suchen der
    nonvar(T),enqu_ol(E,T).  % offenen Restl.

?- X=[_|_],enqu_ol(a,X),
    enqu_ol(b,X),enqu_ol(c,X).
X = [a, b, c|X1].
```

- dequeue:

```
% dequ_ol(-Element,+AlteQueue,-NeueQueue)
dequ_ol(_,[Y|X],_) :-          % Entfernen aus
    var(Y),var(X),fail.        % der leeren Liste
dequ_ol(E,[E|X],[_ |X]) :-     % Entfernen aus
    nonvar(E),var(X).           % der Einerliste
dequ_ol(E,[E|T],T) :-          % Entfernen sonst
    nonvar(T).
```

Offene Listen

- dequeue:

```
?- dequ_ol(E, [b, c | _], N).  
   E = b, N = [c | X1].
```

```
?- dequ_ol(E, [c | _], N).  
   E = c, N = [X1 | X2].
```

```
?- dequ_ol(E, [_ | _], N).  
   false.
```

- **Nachteil:** beim Einfügen nach wie vor Suche nach Listenende erforderlich
→ Differenzlisten

Listendifferenz

jede Liste kann als Differenz zweier Listen dargestellt werden.

?- op(650,xfx,\).

- z.B. Liste $[a, b, c]$ ist äquivalent zu:

$[a, b, c] \setminus []$

$[a, b, c, d] \setminus [d]$

$[a, b, c, d, e] \setminus [d, e]$

$[a, b, c, d, e, f] \setminus [d, e, f]$

- generalisiert zu

$[a, b, c \mid X] \setminus X$

Differenzlisten

- **enqueue:**

```
% enqueue_dl(+Element,+AlteQueue,?NeueQueue)
```

```
enqueue_dl(E,[ ]\[ ],[E|X]\X).
```

```
    % Einfuegen in leere Differenzliste
```

```
enqueue_dl(E,Left\[E|Right],Left\[Right]).
```

```
    % Einfuegen sonst
```

```
% Testdaten:
```

```
[ ]\[ ]           % leere D-Liste
```

```
[a|X]\X           % einelementige D-Liste
```

```
[a,b|X]\X         % zweielementige D-Liste
```

- **enqueue:**

?- $Q = [] \setminus []$, `enqueue_dl(a,Q,N)`.

$Q = [] \setminus []$, $N = [a|X1] \setminus X1$.

?- $Q = [a|X] \setminus X$, `enqueue_dl(b,Q,N)`.

$Q = [a, b|X2] \setminus [b|X2]$, $N = [a, b|X2] \setminus X2$.

?- $Q = [a, b|X] \setminus X$, `enqueue_dl(c,Q,N)`.

$Q = [a, b, c|X3] \setminus [c|X3]$,

$N = [a, b, c|X3] \setminus X3$.

Differenzlisten

- **dequeue:**

```
% dequeue(?Element,+AlteQueue,?NeueQueue).  
dequeue_dl(E,[E|Left]\Right,Left\Right).
```

```
?- Q=[]\[], dequeue_dl(E,Q,N).  
    false.
```

```
?- Q=[a|X]\X, dequeue_dl(E,Q,N).  
    Q = [a|X1]\X1, E = a, N = X1\X1.
```

```
?- Q=[a, b|X]\X, dequeue_dl(E,Q,N).  
    Q = [a, b|X1]\X1, E = a, N = [b|X1]\X1.
```

→ Rechtserweiterung ohne Kopieren und ohne Suche nach dem rechten Listenende

Differenzlisten

- Problemfall:

?- $Q = X \setminus X$, `dequeue_dl(E,Q,N)`.

$Q = [E|X1] \setminus [E|X1]$, $X = [E|X1]$, $N = X1 \setminus [E|X1]$.

Differenzlisten

- Problemfall:

?- $Q = X \setminus X$, `dequeue_dl(E,Q,N)`.

$Q = [E|X1] \setminus [E|X1]$, $X = [E|X1]$, $N = X1 \setminus [E|X1]$.

- Liste mit negativer Länge!

Differenzlisten

→ Entfernen muss blockiert werden, falls Warteschlange leer

```
dequeue2(E,Left\Right,LeftNew\Right) :-  
    Left \=@= Right,  
    Left = [E|LeftNew].
```

```
?- Q=X\X,dequeue2(E,Q,N).  
false.
```

```
?- Q=[a|X]\X,dequeue2(E,Q,N).  
Q = [a|X]\X, E = a, N = X\X.
```

```
?- Q=[a,b|X]\X,dequeue2(E,Q,N).  
Q = [a, b|X]\X, E = a, N = [b|X]\X.
```

```
?- Q=[a,b|X]\[a,b|X],dequeue2(E,Q,N).  
false.
```

Differenzlisten

- Verwendung zur Listenverkettung ohne Dekomposition und Kopieren: `append_d1/3`

```
% append_d1(?DListe1,?DListe2,?DListe3)
append_d1(X\Y,Y\Z,X\Z).
```

$X: \underbrace{a \ b \ c}_{X \setminus Y} \ d \ e \ f \ g \ h \ i \ j \ k$

$Y: \quad \quad \quad \underbrace{d \ e \ f}_{Y \setminus Z} \ g \ h \ i \ j \ k$

$Z: \underbrace{a \ b \ c \ d \ e \ f}_{X \setminus Z} \ g \ h \ i \ j \ k$

- `append_d1/3` ist nur noch ein Fakt!
- drei Unifikationsforderungen über den beteiligten Differenzlisten
- Wo ist die prozedurale Semantik geblieben?
 - konstanter Zeitbedarf?
 - kein Kopieren → ökonomische Freispeicherverwendung

- Testaufrufe:

```
?- L1=[a|X]\X, L2=[a, b|Y]\Y, append_d1(L1,LX,L2).  
    L1 = [a, b|X1]\[b|X1], L2 = [a, b|X1]\X1,  
    LX = [b|X1]\X1.
```

- Testaufrufe:

?- L1=[a|X]\X, L2=[a, b|Y]\Y, append_d1(L1,L2,LX).

L1 = [a, a, b|X1]\[a, b|X1],

L2 = [a, b|X1]\X1,

LX = [a, a, b|X1]\X1.

?- L2=[a, b|Y]\Y, append_d1(LX,L2,L2).

L2 = [a, b|X2]\X2, LX = [a, b|X2]\X3,

LY = X3\X2.

Differenzlisten

- Beispiel: Beseitigen rekursiver Listeneinbettungen: `flatten/2`

```
% flatten mit einfachen Listen (modifizierte Version)
flatten([], []).
flatten(X, [X]) :- atom(X), X\=[ ].
flatten([K|R], F) :-
    flatten(K, KF), flatten(R, RF),
    append(KF, RF, F).
```

```
% flatten(+Liste, ?DListe) mit Differenzlisten
flatten(Xs, Ys) :- flatten_dl(Xs, Ys\[ ]).
flatten_dl([ ], Xs\Xs).
flatten_dl(X, [X|Xs]\Xs) :- atom(X), X\=[ ].
flatten_dl([X|Xs], Ys\Zs) :-
    flatten_dl(X, As\Bs), flatten_dl(Xs, Cs\Ds),
    append_dl(As\Bs, Cs\Ds, Ys\Zs).
```

Entfalten (Unfolding)

Vereinfachen einer Differenzlisten-Klausel durch Integrieren der "statischen" append_dl/3-Definition

```
flatten_dl([X|Xs],Ys\Zs) :-  
    flatten_dl(X,As\Bs),  
    flatten_dl(Xs,Cs\Ds),  
    append_dl(As\Bs,Cs\Ds,Ys\Zs).  
  
%           | |   | |   | |  
% append_dl( X\Y,  Y\Z , X\Z)  
%  
flatten_dl([X|Xs],Ys\Zs) :-  
    flatten_dl(X,Ys\Y1s),  
    flatten_dl(Xs,Y1s\Zs).
```

Entfalten

- Beispiel: Umdrehen einer Liste: `reverse/2`

```
% naives reverse
n_reverse([], []).
n_reverse([X|Xs], R) :-
    n_reverse(Xs, RXs),
    append(RXs, [X], R).

% reverse(+Liste1, ?Liste2)
reverse(Xs, Ys) :- reverse_dl(Xs, Ys\[_]).
reverse_dl([_], Xs\Xs).
reverse_dl([X|Xs], Ys\Zs) :-
    reverse_dl(Xs, As\Bs),
    append_dl(As\Bs, [X|Cs]\Cs, Ys\Zs).
```

- Vereinfachen durch Entfalten

```
reverse_dl([ ],Xs\Xs).
reverse_dl([X|Xs],Ys\Zs) :-
    reverse_dl(Xs,As\Bs),
    append_dl(As\Bs,[X|Cs]\Cs,Ys\Zs).
%           | |      |   |   | |
% append_dl( X\Y,      Y \ Z , X\Z)
%
reverse_dl([X|Xs],Ys\Zs) :-
    reverse_dl(Xs,Ys\[X|Zs]).
```

Entfalten

- Vergleich mit endrekursivem reverse/2

```
reverse(Xs,Ys) :-  
    reverse_dl(Xs,Ys\[ ]).
```

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

```
reverse(Xs,Ys) :-  
    reverse(Xs,[ ],Ys).
```

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

→ Analogie zur Akkumulatorverwendung!

- Sortieren einer Liste: quicksort/2

```
% quicksort(+Liste1,?Liste2)
quicksort([X|Xs],Ys) :-
    split(Xs,X,Vorn,Hinten),
    quicksort(Vorn,Vs),
    quicksort(Hinten,Hs),
    append(Vs,[X|Hs],Ys).
quicksort([ ],[ ]).
```


- Implementation mit Differenzlisten:

```
quicksort(Xs,Ys) :- quicksort_dl(Xs,Ys\[ ]).  
quicksort_dl([X|Xs],Ys\Zs) :-  
    split(Xs,X,Vorn,Hinten),  
    quicksort_dl(Vorn,Ys\[X|Y1s]),  
    quicksort_dl(Hinten,Y1s\Zs).  
quicksort_dl([ ],Xs\Xs).
```

Differenzlisten

- Hauptanwendungsgebiet der Differenzlistentechnik
 - Definite Clause Grammar
- Anwendung der Idee der Differenzlisten
 - Mercury: lineare Ordnung der Teilziele einer Klausel
 - Suchraumverwaltung in Theorembeweisern

→ GWV

Kontextfreie Grammatiken

- Kompositionalität: komplexe Konstituenten entstehen durch *Verkettung* elementarer Konstituenten
 - Regeln:
s \rightarrow **np vp**
 - Übersetzung in ein Logikprogramm:
`s(X) :- np(Y), vp(Z), append(Y,Z,X) .`
 - präterminale Regeln (Wörterbuch):
n \rightarrow **frau**
 - Übersetzung in ein Logikprogramm
`n([frau]) .`

Kontextfreie Grammatiken

- Beispielgrammatik:

`s(X) :- np(Y), vp(Z), append(Y,Z,X).`

`np(X) :- d(Y), n(Z), append(Y,Z,X).`

`vp(X) :- v(X).`

`d([die]).`

`n([frau]).`

`v([liest]).`

Kontextfreie Grammatiken

- Generierung:

```
?- s(X) .                                     #1
s(X1):-np(Y1),vp(Z1),append(Y1,Z1,X1) .      succ(X=X1)
?- np(Y1),vp(Z1),append(Y1,Z1,X1) .          #2
np(X2):-d(Y2),n(Z2),append(Y2,Z2,X2) .      succ(X2=Y1)
?- d(Y2),n(Z2),append(Y2,Z2,X2) .           #3
d([die]) .                                   succ(Y2=[die])
?- n(Z2),append([die],Z2,X2) .               #3a
n([frau]) .                                 succ(Z2=[frau])
?- append([die],[frau],X2) .                 succ(X2=[die,frau])
?- vp(Z1),append(Y1,Z1,X1) .                 #2a
vp(X3):-v(X3) .                             succ(X3=Z1)
?- v(Y3) .                                   #4
v([liest]) .                               succ(X3=[liest])
?- append([die,frau],[liest],X1) .          succ(X1=[die,frau,liest])
X=[die,frau,liest] ;
BT #4
...
BT #1
false.
```

Kontextfreie Grammatiken

- Erkennung:

```
?- s([die,frau,liest]).                                #1
s(X1):-np(Y1),vp(Z1),append(Y1,Z1,X1).                succ(X1=[die,frau,li
?- np(Y1),vp(Z1),append(Y1,Z1,[die,frau,liest])).    #2
np(X2):-d(Y2),n(Z2),append(Y2,Z2,X2).                succ(X2=Y1)
?- d(Y2),n(Z2),append(Y2,Z2,X2).                    #3
d([die]).                                             succ(Y2=[die])
?- n(Z2,append([die],Z2,X2).                         #3a
n([frau]).                                           succ(Z2=[frau])
?- append([die],[frau],X2).                          succ(X2=[die,frau])
?- vp(Z1),append(Y1,Z1,[die,frau,liest])).          #2a
vp(X3):-v(X3).                                       succ(X3=Z1)
?- v(X3).                                           #4
v([liest]).                                         succ(X3=[liest])
?- append([die,frau],[liest],[die,frau,liest]).    succ
true.
```

Kontextfreie Grammatiken

- Analysestrategie:
 - top down, Tiefe zuerst, links-rechts
- blinde Suche:
 - Generieren aller durch die Grammatik lizenzierten Sätze und *anschließender* Vergleich mit dem Eingabesatz
 - generate-and-test ohne Bewertung von Zwischenergebnissen
 - keine Steuerung der Verarbeitung durch die Eingabedaten

Kontextfreie Grammatiken

- Mehrdeutigkeit in der Grammatik

`s(X) :- np(Y), vp(Z), append(Y,Z,X).`

`np(X) :- d(Y), n(Z), append(Y,Z,X).`

`vp(X) :- v(X).`

`d([der]).`

`d([die]).`

`n([mann]).`

`n([frau]).`

`v([lacht]).`

`v([liest]).`

Kontextfreie Grammatiken

```
?- s([die,frau,liest]).
?- np(Y1),vp(Z1),append(Y1,Z1,[die,frau,liest]).
  ?- d(Y2),n(Z2),append(Y2,Z2,X2).
    ?- n(Z2),append(Y2,Z2,X2).
      ?- append([der],[mann],X2).
    ?- vp(Z1),append(Y1,Z1,[die,frau,liest]).
      ?- v(X3).
        ?- append([der,mann],[lacht],[die,frau,liest]). fail
        BT: v(X3)
        ?- append([der,mann],[liest],[die,frau,liest]). fail
        BT: v(X3)
      BT: vp(Z1)
    BT: n(Z2)
  ?- append([der],[frau],X2).
```

Y2=[der]
Z2=[mann]
X2=[der,mann]
X3=[lacht]
X3=[liest]
Z2=[frau]
X2=[der,frau]

Kontextfreie Grammatiken

```
?- vp(Z1),append(Y1,Z1,[die,frau,liest]).
?- v(X3).                                     X3=[lacht]
    ?- append([der,frau],[lacht],[die,frau,liest]). fail
    BT: v(X3)                                X3=[liest]
    ?- append([der,frau],[liest],[die,frau,liest]). fail
    BT: v(X3)
BT: vp(Z1)
    BT: n(Z2)
BT: d(Y2)                                     Y2=[die]
?- n(Z2),append(Y2,Z2,X2).                   Z2=[mann]
    ?- append([die],[mann],X2).              X2=[die,mann]
```

Kontextfreie Grammatiken

```
?- vp(Z1),append(Y1,Z1,[die,frau,liest]).
?- v(X3).                                     X3=[lacht]
?- append([die,mann],[lacht],[die,frau,liest]). fail
BT: v(X3)                                     X3=[liest]
?- append([die,mann],[liest],[die,frau,liest]). fail
BT: v(X3)
BT: vp(Z1)
BT: n(Z2)                                     Z2=[frau]
?- append([die],[frau],X2).                  X2=[die,frau]
?- vp(Z1),append(Y1,Z1,[die,frau,liest]).
?- v(X3).                                     X3=[lacht]
?- append([die,frau],[lacht],[die,frau,liest]). fail
BT: v(X3)                                     X3=[liest]
?- append([die,frau],[liest],[die,frau,liest]). succ

true.
```

Kontextfreie Grammatiken

- Terminierungsprobleme:
 - Aufruf von $np(X)$, $vp(X)$ usw. mit uninstantiierten Variablen
 - keine Terminierung bei rekursiven Regeln

$np(X) \text{ :- } np(Y), pp(Z), append(Y, Z, X) .$

$np(X) \text{ :- } adj(Y), np(Z), append(Y, Z, X) .$

CFG mit Differenzlisten

- Idee: Anwendung von Differenzlisten
 - zur effizienten Verkettung der Teillisten
 - zur Steuerung der Analyse durch Eingabesatz
- Konstituentenprädikate über Differenzlisten

s/1	s(?Liste1\?Liste2)
np/1	np(?Liste1\?Liste2)
...	

CFG mit Differenzlisten

- Interpretation:
 - `Liste1`: zu analysierende Liste, für die die Top-down-Hypothese `s`, `np`, ... überprüft werden soll
 - `Liste2`: Restliste, die von der aktuellen Konstituente `s`, `np`, ... nicht überdeckt wird

CFG mit Differenzlisten

- Regeln:

$s(B \setminus E) :- np(B \setminus M), vp(M \setminus E).$

$np(B \setminus E) :- d(B \setminus M), n(M \setminus E).$

$vp(B \setminus E) :- v(B \setminus E).$

- Präterminale Regeln (Lexikon):

$d([der|R] \setminus R).$

$d([die|R] \setminus R).$

$n([mann|R] \setminus R).$

$n([frau|R] \setminus R).$

$v([lacht|R] \setminus R).$

$v([liest|R] \setminus R).$

CFG mit Differenzlisten

- Erkennung:

```
?- s([die,frau,liest]\[ ]).
    s(B1\E1):-np(B1\M1),vp(M1\E1).

?- np([die,frau,liest]\M1),vp(M1\[ ]).
    np(B2\E2):-d(B2\M2),n(M2\E2).

?- d([die,frau,liest]\M2),n(M2\E2).
    d([der|R1]\R1).
    d([die|R1]\R1).
    ?- n([frau,liest]\E2).
        n([mann|R2]\R2).
        n([frau|R2]\R2).
    ?- vp([liest]\E1).
        vp(B3\E3):-v(B3\E3).
    ?- v([liest]\E3).
        v([lacht|R3],R3).
        v([liest|R3],R3).

true.
```

```
#1
succ(B1=[die,frau,liest],
E1=[ ])
#2
succ(B2=[die,frau,liest],
E2=M1)
#3
fail
succ(R1=[frau,liest]=M2)
#3a
fail
succ(R2=[liest]=E2)
#2a
succ(B3=[liest],E3=E1)
#4
fail
succ(R3=[ ]=E3)
```


CFG mit Differenzlisten

- gesteuerte Suche:
 - Prädikatsaufrufe sind immer mit dem aktuellen Input instanziiert
 - keine freie Generierung von Terminalsymbolketten
 - Suche beschränkt sich auf die Konstituenten, die auf den vorgegebenen Listenanfang passen

CFG mit Differenzlisten

- Terminierungsprobleme nur noch bei linksrekursiven Regeln
 - *das Haus hinter der Straße am Dorfteich*
 - *das Haus hinter der Straße mit dem roten Dach*

`np(B\E) :- np(B\M), pp(M\E) .`

`pp([mit,dem,roten,dach|R]\R) .`

`pp([hinter,der,strasse|R]\R) .`

`pp([am,bach|R]\R) .`

`np([das,haus|R]\R) .`

- Erkennung

```
?- np([das,haus,am,bach]\[ ]).  
    np(B1\E1):-np(B1\M1),pp(M1\E1).  
                                     succ(B1=[das,haus,am,bach],E1=[ ]) )  
?- np([das,haus,am,bach]\M1),pp(M1\[ ]).  
    np(B2\E2):-np(B2\M2),pp(M2\E2).  
                                     succ(B2=[das,haus,am,bach],E2=M1)  
?- np([das,haus,am,bach]\M2),pp(M2\E2).  
    np(B3\E3):-np(B3\M3),pp(M3\E3).  
                                     succ(B3=[das,haus,am,bach],E3=M2)  
.  
.  
.
```

Definite Clause Grammar

- Operatornotation:

-->/2 +Struktur --> +Ziel

- syntaktischer Sonderstatus: Operator wird beim Einlesen sofort in die Differenzlistennotation transformiert

externe DCG-Notation	interne Prolog-Darstellung
s --> np, vp.	s(B,E):-np(B,M),vp(M,E).
v --> [liest].	v([liest E],E).

Definite Clause Grammar

- Grammatik in DCG-Notation

```
s(S) :- s(S, [ ]).
```

```
s --> np, vp.
```

```
np --> d, n.
```

```
vp --> v.
```

```
d --> [die].
```

```
n --> [frau].
```

```
v --> [liest].
```

Definite Clause Grammar

- Syntax:

DCG-Regel ::= Struktur [Terminal] ' -->' Produktion

Produktion ::= Struktur | Terminal |
Produktion (',' | ';') Produktion |
'{' Ziel '}' | '!'

Terminal ::= Liste

- Zusätzliche Ausdrucksmöglichkeiten:

- beliebige Terme als Nichtterminalsymbole
→ Augmentation
- beliebige Listen als Terminalsymbole
- Listen von Termen als Terminalsymbole
- Anreicherung der Grammatikregeln mit zusätzlichen Bedingungen
- Steuerelemente in Grammatikregeln
- rechtsseitig kontextsensitive Regeln

Erweiterungen

- Augmentation (1): Zusätzliche Bedingungen an die Verträglichkeit von (Nicht-) Terminalsymbolen durch Koreferenz
 - Typverträglichkeit
 - Kongruenz, Rektion, ...
- Augmentation (2): Erzeugung von Strukturbeschreibungen auf einem zusätzlichen Argument
- einfachster Fall: Strukturbeschreibung reflektiert die Regelstruktur

$s(S, SB) :- s(SB, S, []).$

$s(s(Snp, Svp)) \rightarrow np(Snp), vp(Svp).$

$np(np(Sd, Sn)) \rightarrow d(Sd), n(Sn).$

$vp(vp(Sv, Snp)) \rightarrow v(Sv), np(Snp).$

$vp(vp(Sv)) \rightarrow v(Sv).$

Implementation von DCG

- Augmentierung auch im Lexikon möglich

```
d(d(die)) --> [die].
```

```
d(d(das)) --> [das].
```

```
n(n(frau)) --> [frau].
```

```
n(n(buch)) --> [buch].
```

```
v(v(liest)) --> [liest].
```

- Testaufruf

```
?- s([die,frau,liest,das,buch],SB).
```

```
    SB = s(np(d(die), n(frau)),  
           vp(v(liest), np(d(das), n(buch))))).
```


- Erzeugung der Strukturbeschreibungen kann auch unabhängig von der Regelstruktur sein
 - linksrekursive Regeln, die eine rechtsrekursive Struktur erzeugen
 - Vermeiden von Terminierungsproblemen
 - Integration von Syntaxanalyse und semantischer Interpretation bzw. Codegenerierung
 - Integration von Syntaxanalyse und strukturellem Transfer

Definite Clause Grammar

- Verwendung des Regelkopfseparators $-->$ auch zur Verkettung sequentieller Prädikatsaufrufe
- "Durchschleifen" von Zwischenergebnissen

$a \text{ --> } b, c, d.$

entspricht

$a(A,B) \text{ :- } b(A,X), c(X,Y), d(Y,B).$

Grammatiken höherer Ordnung

- Spezifikation genereller Transducer
 - String-to-String
 - String-to-Tree
 - Tree-to-String
 - Tree-to-Tree
- Indizierte Grammatiken: Variable können beliebige Strukturen übermitteln
 - Verwendung von Stacks
 - erzeugt verschiedene Instanzen eines Nichtterminalsymbols
 - Grammatiken mit unendlich vielen Nichtterminalsymbolen
 - akzeptiert/generiert Sprachen höherer Ordnung
 - "kontextfreie" Regeln → kontextsensitive (genauer: indizierbare) Sprachen
 - "reguläre" Regeln → kontextfreie Sprachen

Grammatiken höherer Ordnung

- Extremfall Metamorphosis Grammar: linke Regelseite kann beliebige Sequenz aus Nichtterminal- und Terminalsymbolen sein
→ volle Turingmächtigkeit

Grammatikmodellierung

- Beispiel: kontextfreie Sprache mit "regulären" Regeln
 - Sprache: $a^i b^j$ mit $j \geq i$
 - kontextfreie Grammatik

$s \rightarrow s1.$

$s \rightarrow s, b.$

$s1 \rightarrow a, b.$

$s1 \rightarrow a, s1, b.$

$a \rightarrow [a].$

$b \rightarrow [b].$

s

$s\ b$

$s1\ b$

$a\ s1\ b\ b$

$a\ a\ b\ b\ b$

Grammatikmodellierung

- Beispiel: kontextfreie Sprache mit "regulären" Regeln
 - "indizierte" reguläre Grammatik

<code>string(X) --> bs(X).</code>	<code>a string(s(s(0)))</code>
<code>string(X) --> a, string(s(X)).</code>	<code>a a string(s(s(s(0))))</code>
<code>bs(s(X)) --> b, bs(X).</code>	<code>a a bs(s(s(s(0))))</code>
<code>bs(X) --> b, bs(X)</code>	<code>a a b bs(s(s(0)))</code>
<code>bs(0) --> [] .</code>	<code>a a b b bs(s(0))</code>
	<code>a a b b b bs(0)</code>
	<code>a a b b b b bs(0)</code>
	<code>a a b b b b</code>

- konsequente links-rechts-Verarbeitung!

Horn-Logik

- Prolog-Klauseln stellen die einfachste Form prädikatenlogischer Formeln dar
- Klauseln: allquantifizierte Disjunktion von Literalen ohne freie Variable
- Skopus der Quantoren ist die gesamte Klausel

$$A_1 \vee \dots \vee A_n \vee \neg B_1 \vee \dots \vee \neg B_m$$

$$(A_1 \vee \dots \vee A_n) \vee \neg (B_1 \wedge \dots \wedge B_m)$$

$$(A_1 \vee \dots \vee A_n) \leftarrow (B_1 \wedge \dots \wedge B_m)$$

- Horn-Klauseln (definite Klauseln): Klausel mit höchstens einem positiven Literal

$$A \vee \neg B_1 \vee \dots \vee \neg B_m$$

$$A \leftarrow (B_1 \wedge \dots \wedge B_m)$$

- keine Existenzquantoren
können nur über Konstanten simuliert werden (Skolemisierung)

Spezielle Horn-Klauseln

a) mit einem positiven Literal

a1) mit wenigstens einem negativen Literal: Prolog-Regel

$$A \vee \neg B_1 \vee \dots \vee \neg B_m \Leftrightarrow A \leftarrow (B_1 \wedge \dots \wedge B_m)$$

a2) ohne negative Literale: Prolog-Fakt

$$A \Leftrightarrow A \vee \mathbf{F} \Leftrightarrow A \leftarrow \mathbf{T}$$

b) ohne positives Literal

b1) mit wenigstens einem negativen Literal: Ziel

$$\mathbf{F} \vee \neg B_1 \vee \dots \vee \neg B_m \Leftrightarrow \mathbf{F} \leftarrow (B_1 \wedge \dots \wedge B_m)$$

b2) ohne negative Literale: leere Klausel (\square)

$$\mathbf{F} \vee \mathbf{F} \Leftrightarrow \mathbf{T} \leftarrow \mathbf{F}$$

Resolution

- Deduktion: Ableiten von Theoremen aus gegebenen Axiomen
- Übertragung auf die Programmierung:
Unter welchen Bedingungen (mit welchen Variablenbelegungen) kann ein Theorem G (Ziel) aus den gegebenen Axiomen \mathcal{M} (Regeln und Fakten) abgeleitet werden?
→ inverse Zielstellung

Resolution

- Beweisidee:

$$G = P_1 \wedge \dots \wedge P_n$$

$$\neg(\neg P_1 \vee \dots \vee \neg P_n \vee \mathbf{F})$$

$$\neg(P_1 \wedge \dots \wedge P_n \rightarrow \mathbf{F})$$

$$\neg(G \rightarrow \mathbf{F})$$

- Widerlegungsbeweis: $ag(G) \leftrightarrow \neg ef(\neg G)$
- Beweistechnik: $\mathcal{M} \cup \{\neg G\} \models \mathbf{F}$
 - Ableiten der leeren Klausel \square

Resolution

- Resolution: allgemeines Deduktionsverfahren für Klausel-Normalform
- Resolutionsschritt: Zusammenfassen zweier Klauseln, so daß sich positive und negierte Literale gegenseitig aufheben

a) Spezialfall: Widerlegung durch Faktenidentifizierung

$$\begin{array}{c} \neg P \\ P \\ \hline \square \end{array} \qquad \begin{array}{c} \mathbf{F} \leftarrow P \\ P \leftarrow \mathbf{T} \\ \hline \mathbf{F} \leftarrow \mathbf{T} \end{array}$$

b) Resolutionsregel

$$\frac{\begin{array}{c} A_1 \vee \dots \vee A_n \leftarrow B_1 \wedge \dots \wedge B_m \\ C_1 \vee \dots \vee C_k \leftarrow D_1 \wedge \dots \wedge D_l \end{array}}{A_1 \vee \dots \vee A_n \vee C_1 \vee \dots \vee C_k \leftarrow B_1 \wedge \dots \wedge B_m \wedge D_1 \wedge \dots \wedge D_l}$$

Resolution

- Resolutionsregel für Horn-Klauseln

$$\frac{\begin{array}{c} A \leftarrow B_1 \wedge \dots \wedge P \wedge \dots \wedge B_m \\ P \leftarrow D_1 \wedge \dots \wedge D_l \end{array}}{A \leftarrow B_1 \wedge \dots \wedge B_m \wedge D_1 \wedge \dots \wedge D_l}$$

- das Resultat eines Resolutionsschrittes ist wieder eine Horn-Klausel

Resolution

- Auswahl des zu resolvierenden Literals:
 - L-Resolution (linear resolution):
es wird immer die im vorangehenden Resolutionsschritt ermittelte Resolvente weiterverarbeitet
 - SL-Resolution (selective linear resolution):
mit einer Auswahlfunktion wird immer eine resolvierbare Klausel ausgesucht
 - SLD-Resolution (selective linear resolution for definite clauses):
SL-Resolution für Horn-Klauseln

- Kopplung von Resolution und Unifikation

$$\frac{\left. \begin{array}{l} A \leftarrow B_1 \wedge \dots \wedge P_1 \wedge \dots \wedge B_m \\ P_2 \leftarrow D_1 \wedge \dots \wedge D_l \end{array} \right\} \text{ mit } P_2 = \sigma(P_1)}{\sigma(A) \leftarrow \sigma(B_1) \wedge \dots \wedge \sigma(B_m) \wedge \sigma(D_1) \wedge \dots \wedge \sigma(D_l)}$$

Negation

- closed world assumption (CWA): Metaregel zur Inferenz negierter Grundatome

$$\neg(\mathcal{M} \models \mathcal{G}) \rightarrow \mathcal{M} \models_{CWA} \neg \mathcal{G}$$

- negation as finite failure (NFF): Metaregel, die die Negation eines Grundatoms auf das Scheitern seiner Ableitbarkeit in endlich vielen Schritten zurückführt

$$\neg G \begin{cases} \text{fail} & \mathcal{M} \models_{SLD} G \\ \text{success} & \neg(\mathcal{M} \models_{SLD} G) \end{cases}$$

- keine Variablenbindung in negierten Ausdrücken möglich
- "Floundering" für Atome mit nichtinstanzierten Variablen
- SLDNF-Resolution: SLD-Resolution mit NFF-Regel

Logikprogrammierung, wozu?

- Verarbeitungsmodell, Programmierstil und Programmiersprache bedingen einander
- jedoch keine strikten Abhängigkeiten:

Problem →

Verarbeitungsmodell →

Programmierstil →

Programmiersprache

- Verarbeitungsmodell ist primär, Programmiersprache ist sekundär
- Welches Verarbeitungsmodell ist angemessen für ein Problem?

Verarbeitungsmodell

- Klauselarbeitung durch Resolution
 - Nichtdeterminismus
 - Ersetzungsregeln
 - Rekursion
- Unifikation
 - Pattern-Matching für Termstrukturen
 - Listenverarbeitung
- Eignung für komplex strukturierte Symbolverarbeitungsprobleme
- Verarbeitungsmodell ist sehr flexibel
 - veränderte Resolutionsstrategien
 - erweiterte Unifikation

- relationale Programmierung
 - Ergebnismengen statt Einzelergebnisse
 - Richtungsunabhängigkeit der Berechnung
 - Betonung rekursiver Problemlösungen
 - im Kernbereich deklarativ: referentielle Transparenz ist gewahrt
 - Zurückdrängen prozeduraler Aspekte: höherer Abstraktionsgrad
 - nichtdeklarative Komponenten in isolierten Bereichen (Arithmetik, Suchraummanipulation, extralogische Prädikate)

Programmierstil

- relationale Programmierung als Ideal?
 - simples Skopusmodell
 - Seiteneffekte als Fehlerquellen zurückgedrängt
- aber:
 - nichtrelationaler Charakter der Arithmetik
 - hoher Abstraktionsgrad erschwert Verständlichkeit
 - relativ schlechte Kompatibilität mit objektorientierter Programmierung

Programmierstil

- Prolog erlaubt unterschiedliche Programmierstile:
- funktionale Programmierung:
 - funktionale Auswertung in der Arithmetik
 - eindeutige Berechnungsergebnisse: `cut`
 - Simulation der Funktionskomposition durch Sequenzen von Prädikatsaufrufen
- zustandsorientierte (imperative) Programmierung:
 - Klauseln als Speicherzellen: `assert/retract`
 - nichtmonotone Steuerkonstrukte: `cut`
- objektorientierte Programmierung:
 - Informationskapselung: über das Modulsystem
 - Vererbung: im Bereich der statischen Wissensrepräsentation
 - schwache Kopplung mit einem Objektsystem: XPCE

Programmiersprache

- Verarbeitungsmodell der Logikprogrammierung enthält im Kern bereits sehr leistungsfähige Basiskomponenten (Suche, Unifikation)
- flexible Syntax
- flexible Semantik
- gut geeignet für Experimente mit neuen Verarbeitungsmodellen
→ rapid prototyping für Funktionsprototypen

Erweiterungen

- Integration funktionaler Programmierung
- Strikt deklarative Logikprogrammierung
- Constraint-Lösen
- Deduktive Datenbanken
- Graphunifikation
- Induktive Logikprogrammierung

Integration funktionaler Programmierung

- Problem: funktionale Auswertung zur Zeit nur in spezieller Umgebung
 - extralogisches Konstrukt
 - keine relationale Auswertung möglich
- Ziel: vollständige Integration in die Unifikation
- prinzipieller Lösungsansatz: erweiterte Unifikation für zwei Atome A und B
 - Disagreement-Menge D :
enthält die nicht unifizierbaren Bestandteile von A und B , wenn diese als Funktionensymbol definiert sind

$$p(+ (2, x)) \sqcup p(+ (3, y)) \Rightarrow D = + (2, x), + (3, y)$$

- funktionale Evaluation der Elemente der Disagreement-Menge

Strikt richtungsunabhängige Logikprogrammierung

- Lösungsidee: Klauseln werden nicht mehr interpretativ abgearbeitet
 - Deklaration von Verarbeitungsmodi (Instanziierungsvarianten) und Determinismus-Modi
 - *Compilation* in ausführbaren Code für die verschiedenen Aufrufvarianten
 - sequentielle Abhängigkeiten werden separat verwaltet
 - → Mercury (University of Melbourne)
- Deklaration der Verarbeitungsmodi

```
:- pred append(list(T), list(T), list(T)).  
:- mode append(in, in, out) is det.  
:- mode append(in, out, in) is semidet.  
:- mode append(out, in, in) is semidet.  
:- mode append(out, out, in) is multi.
```


Strikt richtungsunabhängige Logikprogrammierung

- Problem: Reihenfolgeabhängigkeit bei seiteneffektbehafteten Teilzielen (z.B. Input/Output)
- Lösungsansatz: explizite Verwaltung von Berechnungszuständen (state of the world)
 - Hinzufügen von zwei Zustandsparametern zu jedem IO-Prädikat
 - explizite Modellierung von Sequenzbeziehungen
 - analog zur Differenzlistentechnik

Constraint-Lösen

- Problem: Instanziierung von Teilzielen über großen Faktenmengen führt zu ineffizientem Code: trial-and-error
- Ziel: möglichst frühzeitige Bedingungsüberprüfung schon bei der Variablenbindung und nicht erst bei der Auswertung des Prozedurkörpers

```
?- X in 1 ..5, Y in 3 .. 7, X?>Y,  
   indomain(X).  
X = 4, Y = 3 ;  
X = 5, Y = 3 ;  
X = 5, Y = 4.
```

Constraint-Lösen

- Lösungsansatz: Constraint-Satisfaction
- effiziente Lösungen bekannt, aber nur für Teildomänen
 - Gleitkommazahlen , Rationalzahlen: Intervallarithmetik, lineare Gleichungen/Ungleichungen
 - Ganze Zahlen: Intervalle, Folgen, Aufzählungen, Gleichungen, Ungleichungen, negierte Gleichungen
 - Spezialfälle: endliche symbolische Domänen, Boolesche Domänen

Constraint-Lösen

- 1. Schritt: Erweiterte relationale Programmierung
 - auch für numerische Aufgaben

```
peano(0,0).  
peano(X,s(Y)):-  
    X?>0,  
    peano(X-1,Y).
```

- 2. Schritt: Constraint Logic Programming
 - Constraint-Gleichungen sind Resultat der Unifikation
 - Gleichungen sind a priori nicht unbedingt vollständig bekannt
 - werden erst über mehrere Unifikationen hinweg angereichert

Deduktive Datenbanken

- Problem 1: Prädikatenkalkül ist nur semi-entscheidbar
→ Terminierungsprobleme
- Ziel: Terminierungssicherheit auch für Datenbanken mit Inferenzfähigkeit
- Lösungsansatz:
 - faktengesteuerte Inferenztechniken (OLDT-Resolution, magic sets)
 - Vereinfachung: keine komplexen Terme an Argumentpositionen (keine Funktionensymbole)
- DATALOG:
 - keine Rekursion über Datenstrukturen (nur über der Datenbasis)
 - vereinfachte Unifikation
 - effiziente Indexierung

Deduktive Datenbanken

- Problem 2: unvollständige Behandlung der Negation
- Ziel: Variablenbindungen für negierte Anfragen

```
amount(i_134,2487.28).  
amount(i_147,34.00).  
amount(i_165,1835.90).  
recipient(i_134,meier).  
recipient(i_147,schulze).  
recipient(i_165,mueller).  
paid(i_147).
```

```
?- not(paid(Inv)), amount(Inv,Amount),  
    recipient(Inv,To).
```

- Lösungsansatz: Sortenbeschränkung für die Argumente

```
invoice(i_134).  
invoice(i_147).  
invoice(i_165).
```

```
?- invoice(Inv), not(paid(Inv)),  
   amount(Inv,Amount), recipient(Inv,To).
```

```
I=i_134, Amount=2487.28, To=meier ;
```

```
I=i_165, Amount=1835.90, To=mueллер.
```

- a) direkter Einbau von Sortenbeschränkungen in die Unifikation (sortierte Logik)
- b) explizite Einschränkung der zulässigen Sorten an den Argumentstellen

```
amount(X::invoice,Y::amount_of_money)  
recipient(X::invoice,Y::company)  
paid(X::invoice)
```

- Problem 3: Integritätsconstraints (Gleichungen und Ungleichungen)

Graphunifikation

- Problem: Stelligkeit ist Bestandteil der Prädikatsdefinition
→ unübersichtliche Programme bei stark unterspezifizierter Modellierung (z.B. Grammatiken für natürliche Sprache)
- Ziel: Beschränkung auf die Angabe der jeweils relevanten Information
→ Merkmalstrukturen sind "seitlich erweiterbar"

$$\left[\begin{array}{cc} a & \boxed{1} \\ b & \boxed{1} \end{array} \left[\begin{array}{cc} c & \boxed{2} \\ d & \boxed{2} \end{array} \right] \right]$$

$$\left[a \quad \boxed{1} \left[b \quad \boxed{1} \right] \right]$$

Graphunifikation

- Subsumtion: A subsumiert B gdw. jeder Pfad aus A auch in B enthalten ist
- Unifikation: der Unifikator von A und B ist die allgemeinste Merkmalstruktur, die sowohl von A als auch von B subsumiert wird

$$\left[\begin{array}{cc} \text{cat} & n \\ \text{agr} & \left[\begin{array}{cc} \text{cas} & \text{nom} \end{array} \right] \end{array} \right] \sqcup \left[\begin{array}{cc} \text{cat} & n \\ \text{agr} & \left[\begin{array}{cc} \text{gend} & \text{fem} \\ \text{num} & \text{sg} \end{array} \right] \end{array} \right] = \left[\begin{array}{cc} \text{cat} & n \\ \text{agr} & \left[\begin{array}{cc} \text{cas} & \text{nom} \\ \text{gend} & \text{fem} \\ \text{num} & \text{sg} \end{array} \right] \end{array} \right]$$

- Implementation von DAGs als offene Listen

```
:- op(500, xfy, :).
```

```
fs1([cat : n,  
     agr : [cas : nom | _]  
     | _]).
```

```
fs2([cat : n,  
     agr : [gend : fem,  
           num : sg | _]  
     | _]).
```

- Implementation der Unifikation

```
unify(Dag,Dag) :- !.  
    % identische DAGs unifizieren  
  
unify(Path:Value|Dags1],Dag) :-  
    % Dag1:Path:Value unifiziert  
    pathval(Dag,Path,Value,RemDags),  
    % mit Dag2:Pfad:Value  
    unify(Dags1,RemDags).  
    % Rest-DAGs unifizieren
```

Graphunifikation

```
pathval(Dag1,Feature:Path,Value,Dags) :- !,  
    % Pfad ist Sequenz aus Merkmalen  
    pathval(Dag1, Feature, Dag2, Dags),  
    % Zerlegung des Pfades: Teil 1  
    pathval(Dag2, Path, Value, _).  
    % Zerlegung des Pfades: Teil 2  
  
pathval([Feature:Value1|Dags], Feature, Value2, Dags) :- !,  
    % relevantes Merkmal  
    unify(Value1,Value2).  
    % gefunden und unifiziert  
  
pathval([Dag|Dags1],Feature,Value,[Dag|Dags2]) :-  
    % Rekursiver Abbau des Graphen  
    pathval(Dags1,Feature,Value,Dags2).
```


Induktive Logikprogrammierung

- Logikprogramme werden aus Beispieldaten und Hintergrundwissen generalisiert
- Kombination mit Techniken des Maschinellen Lernens
- Beispieldaten, Hintergrundwissen und Lernresultate werden als Logikprogramme beschrieben
- Anwendungen
 - Fehlerdiagnoseregeln
 - Struktur-Wirkungs-Regeln für Medikamente
 - Regeln zur Vorhersage der Sekundärstruktur von Proteinen