Logik-Programmierung 2

School of Engineering

- Praktische Umsetzungen des Resolutionsbeweises
- Hornklauseln
- Prolog

Wieso Prolog lernen?

Die 4 (5) grundlegenden Programmierparadigmen

- Imperativ (1956, **Fortran**)
- Funktional (1959, **Lisp**)
- Objektorientiert (1972, **Smalltalk**)
- Logisch (1974, **Prolog**)
- Deklarativ (1975, **SGML**) umstritten, da kein eigentlicher Programmfluss.

- Prolog ist der erste Repräsentant eines grundlegenden Paradigmas
- Alle symbolischen KI Systeme sind durch Prolog beeinflusst

- Jeder Informatiker der sich mit symbolischer AI befassen möchte, sollte Prolog bzw. das Paradigma der logischen Programmierung kennen.

Prolog, Datalog und SQL

- Prolog = Programming + Logic
  - Prolog ist eine Programmiersprache für Logik
  - Verwendung hauptsächlich in Europa und Japan u.a. japanischen 5th Generation Project

- Datalog = Databases + Logic
  - Datenbank-Programmiersprache für deduktive Datenbanken
  - Syntaktisch und semantisch ähnlich zu Prolog
  - Einfache Version von Prolog
    - *einige Features von Prolog wurden weggelassen*
  - Fokus auf Ergänzung von Datenbanken durch deduktive Fähigkeiten

	SQL	Datalog	Prolog
recursive data	?		+
recursive queries		+	+
Horn clauses	+	+	+
control flow			+
functions	+		
typed values	+		
named values	+		

Praktische Umsetzung des Resolutionsbeweises

Problem

- Wir haben gesehen:
  - Prädikatenlogik 1. Ordnung (PL1) ist gut geeignet um Wissen zu modellieren
  - Das Resolutionskalkül für PL ermöglicht es Klauseln "mechanisch", d.h. mit einem Programm zu verifizieren.

- Aber es gibt sehr viele verschiedene Möglichkeiten
  - Resolvente mit komplementären Literale zu kombinieren.
  - Unifikationen vorzunehmen

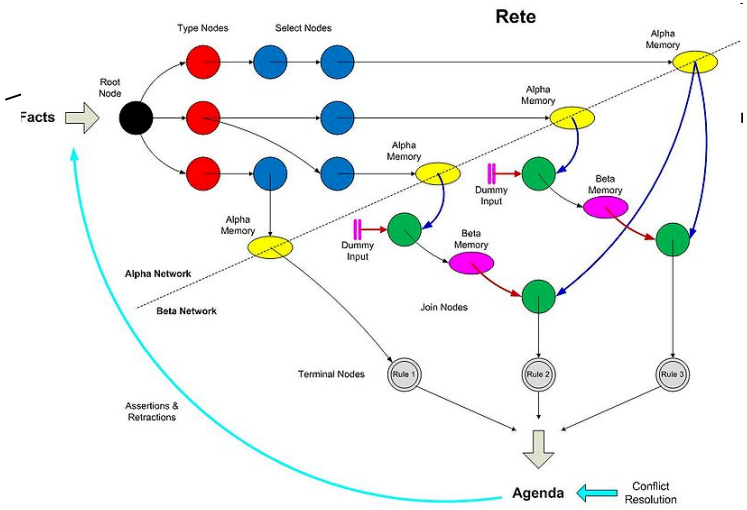
- Ab einer gewissen Anzahl von Klauseln wird der Resolutionskalkül für PL1 sehr ineffizient.

> Optimierung des Beweisverfahrens ist notwendig

Massnahmen zur Effizienzsteigerung

Ansatz 1: Entwicklung von Heuristiken zur Klauselauswahl

- Idee: man betrachtet die Klauselauswahl als Suchproblem und setzt „intelligente“ Suchverfahren ein
  - Beispiele für mögliche Strategien:
    - bevorzuge möglichst kurze Klauseln
    - entferne Klauseln, die von anderen „subsumiert“ werden.
  - Schnellere Auswahl von Regeln mit teilweise gemeinsamen Prämissen mit Hilfe eines Baum Datenstruktur -> Rete Algorithmus
  - http://en.wikipedia.org/wiki/Rete_algorithm#Reference



Massnahmen zur Effizienzsteigerung

Ansatz 2: Einschränkung der Ausdrucksmächtigkeit

- Idee: Für viele Anwendungen reicht eine Ausdrucksmächtigkeit, die zwischen Aussagenlogik und PL1 liegt.

=> Einschränkung auf eine weniger ausdrucksmächtige Sprache, z.B.:

- nur all-quantifizierte Variablen
- keine Disjunktion (oder-Verknüpfung) in Termen

In der Praxis oft:

- Einschränkung auf Spezialform, die sog. **Hornklauseln** (benannt nach dem Logiker Alfred Horn).

-> Klauseln ***mit höchstens einem positiven*** Literal

Hornklauseln

Horn Klausel

- Klausel, auch Disjunktionsterm genannt, versteht man die Disjunktion von Literalen:

$$\Phi_1 \vee \Phi_2 \vee \Phi_3 \ldots \vee \Phi_n$$

- Φ_i ist ein automarer Ausdruck oder ein negierter atomarer Ausdruck.

- Eine Horn-Klausel ist eine Klausel mit höchstens einem positiven Literal, d.h. höchstens einem Literal, das nicht negiert ist.

$$\neg p \vee \neg v \vee \ldots \vee \neg t \vee u$$

Darstellungsformen von Hornklauseln

- Horn-Klauseln lassen sich nach den Regeln der Aussagenlogik auch als Implikationen darstellen

$$\neg\Phi \vee \Psi = \Phi \Rightarrow \Psi$$

- Eine Horn-Klausel kann genau eines oder gar kein nicht-negiertes Atom enthalten

- Es kann also auch vorkommen, dass gar keine Literale mit negierten atomaren Ausdrücken auftreten.

- Es gibt also drei Grundtypen von Horn-Klauseln.

1. Definite Hornklausel = Regel

- Genau ein positives Literal: $\neg x_1 \vee \ldots \vee \neg x_n \vee y$
- Als Implikation dargestellt: $x_1 \wedge \ldots \wedge x_n \Rightarrow y$
- In Worten: Wenn $x_1, \ldots, x_n$ wahr sind, dann ist auch y wahr

Konventionen

- Man kehrt den Pfeil meistens um: $P \leftarrow P_1 \wedge P_2 \wedge \ldots \wedge P_n$
- Das positive Literal P bezeichnet man als ***Konklusion***
- Die negativen Literale $\neg P_1, \neg P_2, \ldots, \neg P_n$ heissen ***Prämissen***

2. Faktenklausel = Fakt

- Keine negativen Literale : y
- Als Implikation dargestellt: $1 \Rightarrow y$
- In Worten: y ist wahr

Konventionen

- Man kehrt den Pfeil meistens um: $P \leftarrow _$ oder $P \vee _$
- bzw. lässt die Prämisse weg: P

3. Zielklausel = Anfrage

- Kein positives Literal: $\neg x_1 \vee \ldots \vee \neg x_n$
- Als Implikation dargestellt: $x_1 \wedge \ldots \wedge x_n \Rightarrow 0$
- In Worten: $x_1, \ldots, x_n$ sind nicht alle wahr

Konventionen

- Man kehrt den Pfeil meistens um: $_ \leftarrow P_1 \wedge P_2 \wedge \ldots \wedge P_n$

- *Anfrage: folgt aus dem gegebenen Wissen die Gültigkeit von* $P_1 \wedge P_2 \wedge \ldots \wedge P_n$
- Widerspruchsbeweis : wenn obige Aussage aus dem gegebenen Wissen nicht hergeleitet werden kann dann müssen P_i wahr sein

Beantwortung von Anfragen

Gegeben:

- Wissensbasis mit
  - i) Fakten der Form: **P** (in Hornklauselschreibweise: **P** ← _)
  - ii) Regeln der Form:
    H ∨¬**B**$_1$∨ ¬**B**$_2$∨ **...**∨ ¬**B**$_m$ (Hornklauselschreibweise: **H** ← **B**$_1$∧**B**$_2$∧ **...** ∧**B**$_m$)

Anfrage:

- Zu beweisen ist die Gültigkeit einer Aussage: **P**$_1$∧**P**$_2$∧ **...** ∧**P**$_n$

Ansatz:

- Mit Widerspruchsbeweis (Resolution), wobei man von der negierten Aussage ausgeht:

  ¬**(P**$_1$∧**P**$_2$∧ **...** ∧**P**$_n$) ist äquivalent zu

  ¬**P**$_1$∨ ¬**P**$_2$∨ **...** ∨ ¬**P**$_n$ (in Hornklauselschreibweise: _← **P**$_1$∧**P**$_2$∧ **...** ∧**P**$_n$)

- Der Widerspruch ist dann aufgedeckt, wenn man die „leere Klausel“ _ ← _ ableiten kann.

Inferenzregel für Hornklauseln

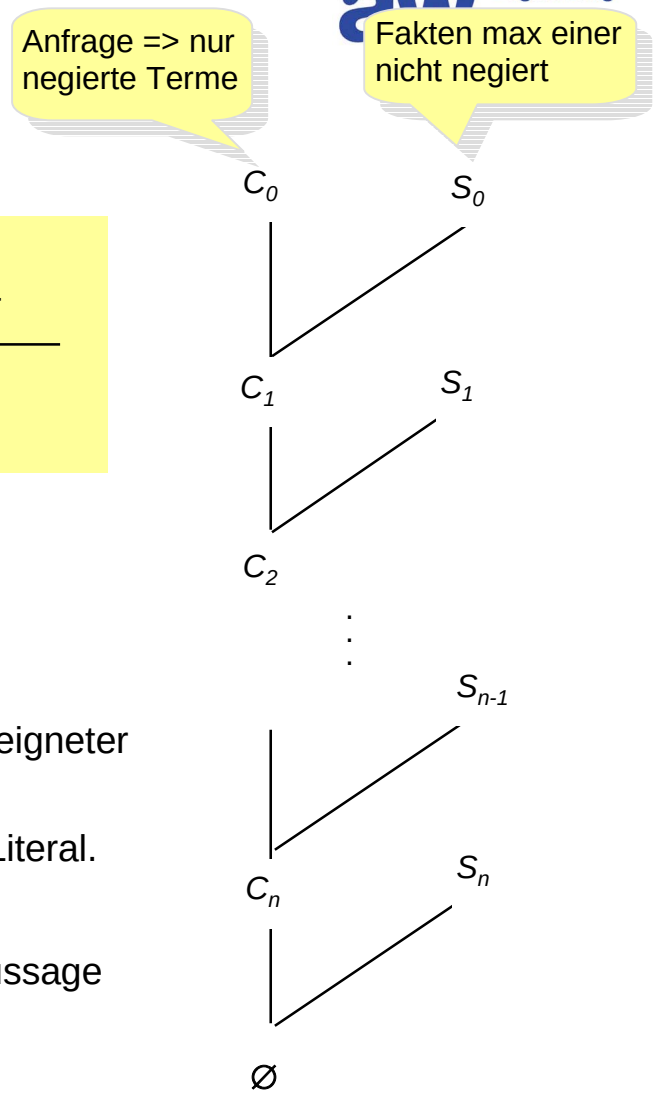
- **Inferenzregel (OLD-Resolution)**

H_1 mit **H** unifizierbar:
$H_1\sigma = H\sigma$

$$\frac{\{\neg H_1, \neg H_2, \ldots, \neg H_m\} \quad , \quad \{H, \neg B_1, \neg B_2, \ldots, \neg B_n\}}{\{\neg H_2, \ldots, \neg H_m, \neg B_1, \neg B_2, \ldots, \neg B_n\}\,\sigma}$$

- **Ordered-Linear Resolution for Definite Clauses**
  - starte mit negierter Anfrage als einer Klausel C_0
  - Lineare Resolution: nimm im nächsten Resolutionsschritt die Resolvente C_i als eine der beiden Klauseln und resolviere mit geeigneter Klausel S_i
  - Literale einer Klausel sind geordnet, resolviere mit dem ersten Literal.

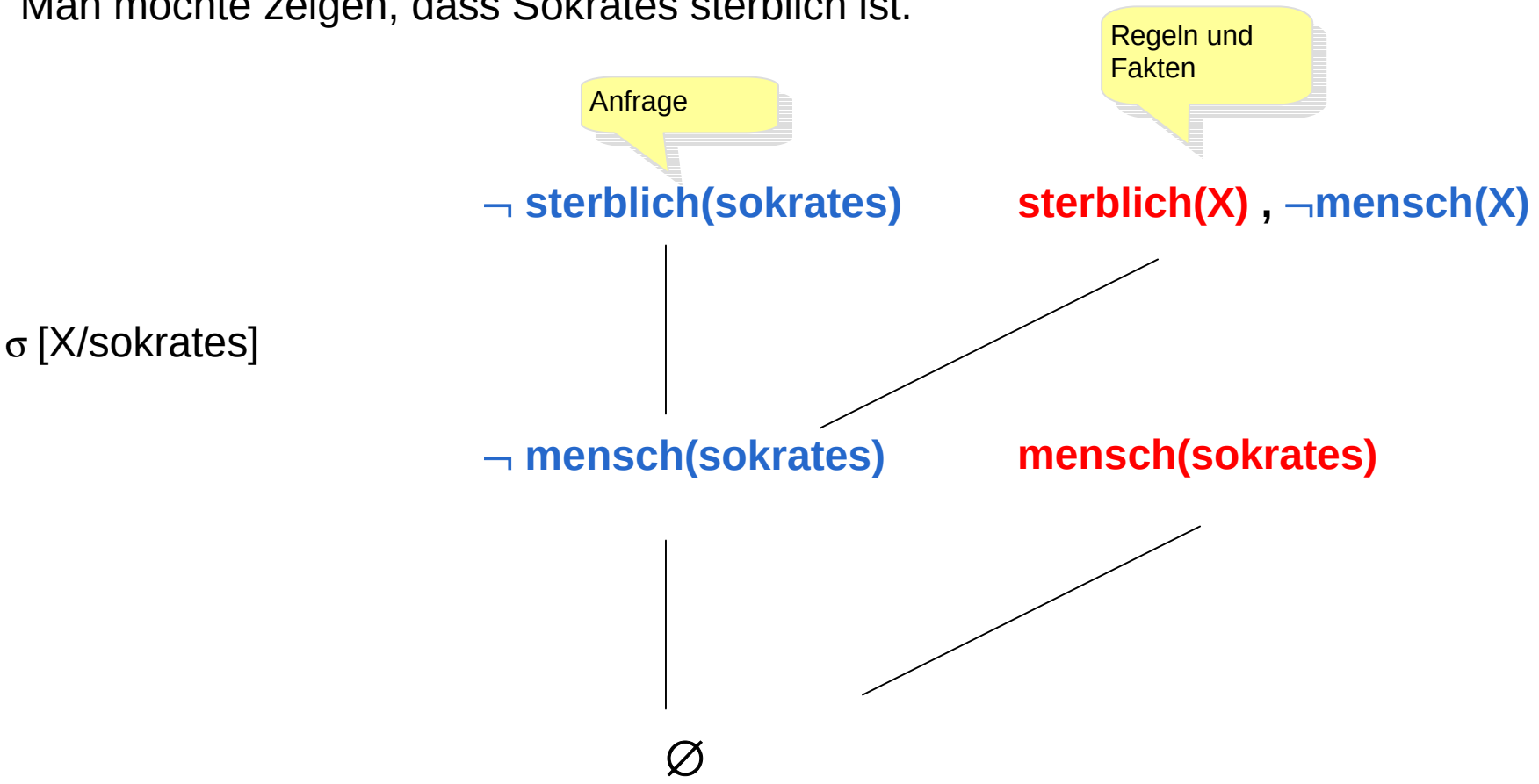
=>in jedem Schritt wird ein negatives Literal der zu beweisenden Aussage abgearbeitet.
Stop bei leerer Resolvente



C_i: Center Clauses (Zentrumsklauseln)
S_i: Side Clauses (Seitenklauseln)

Beispiel 1

- Man möchte zeigen, dass Sokrates sterblich ist.



OLD-Resolution

- Ein Resolutionsschritt ist durchführbar, wenn die *Konklusion* der Regel (= nicht negierter Teil) mit der Anfrage unifiziert werden kann:
  - Variablen werden mit Konstanten belegt
  - eine Variable muss bei jedem Vorkommen den gleichen Wert haben.
- Enthält die Anfrage Variablen, so werden "gültige Werte" der Variablen eingesetzt (Konstanten oder einheitliche Benennung von Variablen)
- Es werden alle möglichen Belegungen (=Unifikationen) durchprobiert
- =>Rücksetzen (**Backtracking**) und mit **alternativer Belegung** für die zu unifizierenden Variablen.

  Beispiel: Regelbasis

  p(3).
  p(4).
  r(X,Y) ← p(X), X = Y.

  Anfrage: r(X,4) liefert zuerst X = 3 keine Lösung, dann Versuch mit X = 4 -> Lösung

- Falls für eine Anfrage keine für Resolution anwendbare Regeln oder Fakten mehr gefunden werden, ist diese nicht beweisbar bzw falsch (vgl. Closed-World Assumption)

Beispiel 2

- Eine Hornklausel-Wissensbasis besteht aus einer Menge von Regeln und Fakten.

```
human(john)
human(paul)
human(mary)
male(john)
male(paul)
female(mary)
father(john,mary)
mother(mary,paul)

man(X) ← human(X) ∧ male(X)
woman(X) ← human(X) ∧ female(X)
parent(X,Y) ← mother(X,Y)
parent(X,Y) ← father(X,Y)
ancestor(X,Y) ← parent(X,Y)
ancestor(X,Y) ← parent(X,Z)∧ ancestor(Z,Y)
```

Fakten

(Faktenbasis)

Regeln

(Regelbasis)

Beispiel 2

- Beispiele für mögliche Anfragen:

← `human(john)`			
	Resultat:	`yes`	;;; Wahrheitswert
← `female(john)`			
	Resultat:	`no`	
← `female(peter)`			
	Resultat:	`no`	
← `human(X)`			
	Resultat:	`X = john` oder	;;; Belegungen (Unifikation),
		`X = paul` oder	;;; für die Anfrage wahr
		`X = mary`	;;; wird
← `man(john)`			
	Resultat:	`yes`	
← `man(X)`			
	Resultat:	`X = john` oder	
		`X = paul`	
← `parent(mary,X)`			
	Resultat:	`X = paul`	
← `ancestor(X,Y) ∧ male(X)`			
	Resultat:	`X = john, Y = mary` oder	
		`X = john, Y = paul`	

Zusammenfassung: Hornklauseln

Pro

- Mit Hornklauseln lassen sich Wissensbasen aufbauen, mittels
  - Fakten (zur Repräsentation von Objekten und Sachverhalten)
  - Regeln (zur Repräsentation von Beziehungen (Abhängigkeiten, Kausalitäten, Strukturbeziehungen, usw.).
  - Variablen in Regeln (und Fakten) erlaubt die Abstraktion von konkreten Werten (Vorteil gegenüber der Aussagenlogik).

- Anfragen an die Wissensbasis werden als spezielle Klauseln formuliert und mittels eines linearen Resolutionsbeweises beantwortet (jedoch müssen alle möglichen Unifikationen durchprobiert werden).

Kontra

- Weniger ausdrucksstark als Prädikatenlogik 1. Stufe
  - keine alternativen in Prämissen der Form $C \leftarrow A \vee B$
  - keine negierten Prämissen (bzw. Anfragen der Form: $\leftarrow \neg C$)

In Prolog gelten diese Einschränkungen nicht

PROLOG

Anwendung der Hornklausel-Logik

Hornklausellogik

- Grundlage für Sprachen zur logischen Programmierung (insbesondere Prolog)
- Grundlage deduktiver Datenbanken

PROLOG: PROgramming in **LOG**ic

- Entwickelt 1972 von Philippe Roussel in der Universität von Marseille
- basiert auf Hornklauseln und (linearem) Resolutionsbeweis:
  - (Genauer: **SLD**-Resolution: **L**inear resolution with **S**election function for **D**efinite clauses)

ähnlich zu OLD

- Existenzquantor entfällt, Allquantor wird implizit vorausgesetzt.
- Terme als Datenstrukturen
  - Variablen
  - Konstante
  - Funktionssymbol mit Argumentliste (wiederum Terme)

Prolog Syntax

Syntaktische Besonderheiten:

- $\wedge$ wird zu ,
- $\leftarrow$ wird zu :-
- jede Klausel wird mit einem Punkt "." beendet

- **Notation von Regeln:**
  - Struktur: H :- B_1, B_2, ... ,B_m „H gilt, wenn $B_1 \wedge B_2 \wedge \ldots \wedge B_m$ „ gilt
  - **Beispiel:** *oma(X,Y)* :- *mutter(X,Z)* , *mutter(Z,Y)* .

- **Notation von Fakten:**
  - Struktur: D :- Hornklausel ohne negatives Literal
  - **Beispiel:** *oma(emma, klaus)* :- (meist ohne :-)

- **Notation von Anfragen:**
  - Struktur: :- A Hornklausel ohne positives Literal
  - **Beispiel:** *:- oma(X, klaus).*

… Prolog Syntax

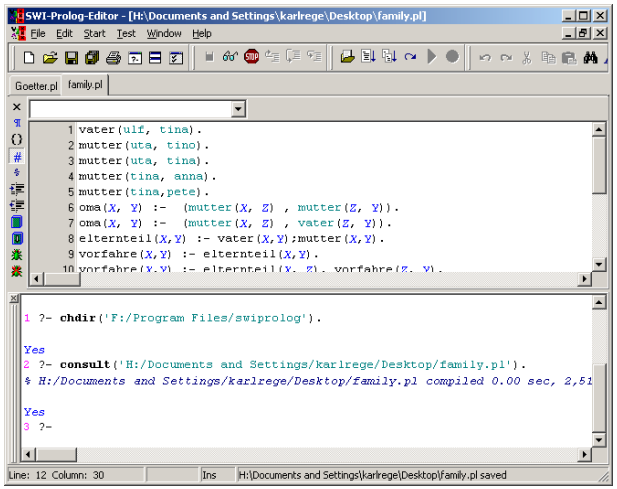
- **Terme:**
  - Argumente eines Literals sind Terme
  - Terme sind entweder Variablen, Zahlen, Konstanten oder Funktionen

- **Konventionen** (d.h. in der Prolog-Programmierung übliche Festlegungen)
  - **Variablen** werden durch einen **Grossbuchstaben** bezeichnet oder beginnen mit "_"
    z.B.: **X** oder **_X**
  - **Konstanten**, **Funktionen** und **Prädikatsnamen** beginnen mit einem **Kleinbuchstaben**.
    z.B.: **vater_von(hans, V)**
  - bei **Funktionen** werden meist Variablen für Eingabe- und Ausgabeparameter verwendet.
    z.B. **sin(X, Y)** (entspricht Y := sin(X))

- **Datentypen**
  - **Atome :** beliebige Folge von Zeichen
    - beginnen klein und haben keine Leerzeichen: alpha, noun, integral
    - oder sind in ' eingeschlossen: 'Karl Rege', ','
  - **Zahlen**: Ganzzahlen

Prolog Bibliothek /System

- PROLOG stellt vordefinierte Prädikate und Funktionen zur Verfügung, u.a.:
  - Ein- / Ausgabe
  - Arithmetik
  - Listenverarbeitung (z.B., append, member, remove)
  - Konstrukte zur Steuerung der Verarbeitung (cut Operator)
  - je nach Implementierung diverse weitere Ergänzungen



SWI-Prolog
University of Amsterdam
http://www.swi-prolog.org
mit dazugehörigem Editor

Arbeiten mit Prolog Beispiel 1

Gegeben Fakten und Regeln

```
god(zeus).                % facts who is a god
god(apollo).
human(socrates).          % facts about who is human
human(aristotle).
human(plato).
mortal(X) :- human(X).    % a logical assertion (rule) X is mortal if X is human
```

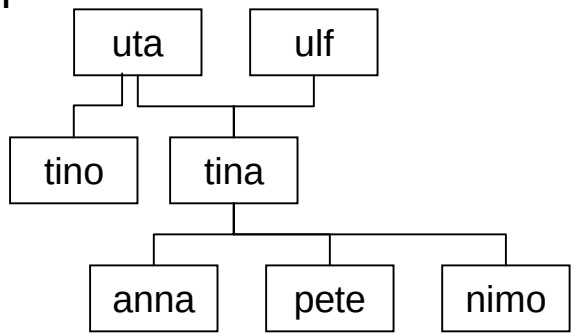
Anfragen erfolgen durch direkte Eingabe von zu überprüfenden Fakten

```
?- human(socrates).       % is socrates a human?
true.


?- god(X).              % who is a god?
X = zeus ;
X = apollo.
```

Arbeiten mit Prolog Beispiel 2

Gegeben Fakten und Regeln für kleinen Stammbaum



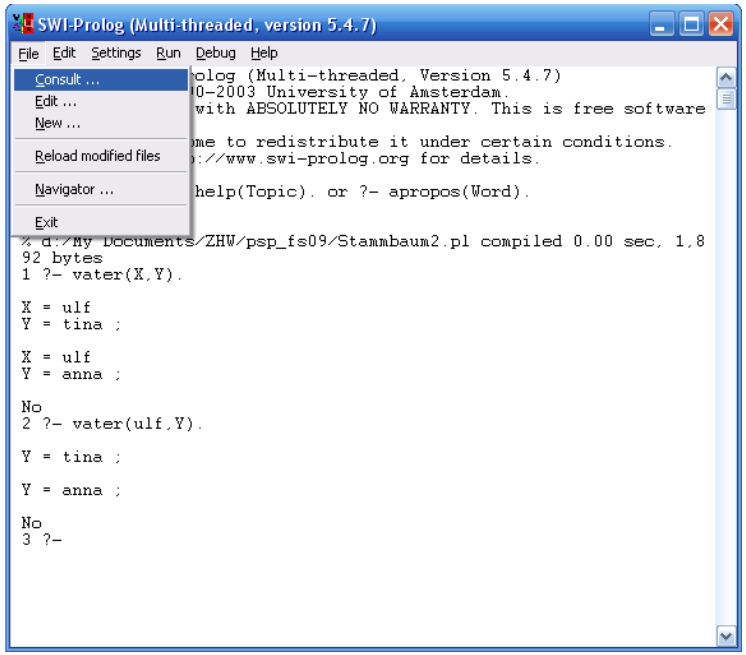

```
vater(ulf, tina).
mutter(uta, tino).
mutter(uta, tina).
mutter(tina, anna).
mutter(tina, pete).
oma(X,Y) :- mutter(X,Z) , mutter(Z,Y).
```

Anfragen erfolgen durch direkte Eingabe von zu überprüfenden Fakten oder Termen mit Variablen, z.B.:

```
?- mutter(uta,tina) .   „Gilt, dass uta die Mutter von Tina ist?“
?- mutter(uta,X) .      „uta ist Mutter von ?“
?- oma(O, anna) .    „Wer ist die  Oma von anna?“
```

Arbeiten mit Kommandozeile

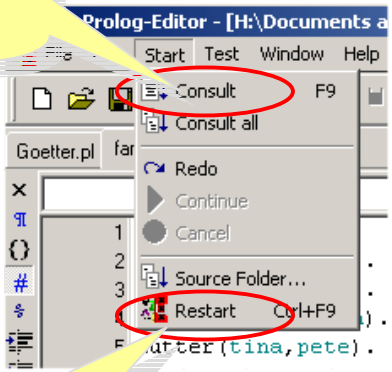
- Erstellen einer Prolog Datei mit z.B. Notepad (.pl)
- starten mit plwin
- laden von *.pl File mittels `consult('<filename.pl>')`
- Anfragen direkt eingeben



Arbeiten mit SWI Editor

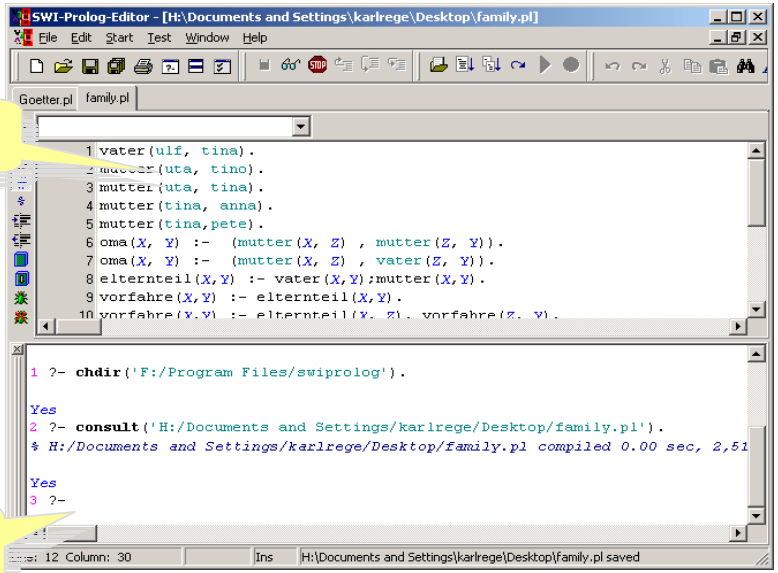
- File Open in Editor
- Laden einer Prolog Datei mit Start -> Consult
- Restart für Neuinitialisierung

füge Inhalt der Datei zur Faktenbasis hinzu



lösche Faktenbasis

Editor



interaktive Konsole

Beantwortung von Anfragen in Prolog

- Die von Prolog durchgeführte Tiefensuche mit Backtracking lässt sich nutzen, um alle möglichen Variabelenbelegungen zu finden, für die eine Anfrage erfüllbar ist.

- Eingabe von Anfrage (mit "." abgeschlossen)
- ";" sucht nach nächster gültigen Unifikation.
  Antwort "No", falls es keine
  weitere Unifikationen gibt.

```
4 ?- oma(X,klaus).

No
5 ?- vater(X,tina).

X = ulf ;

No
6 ?- mutter(uta,X).

X = tino ;;

X = tina ;;

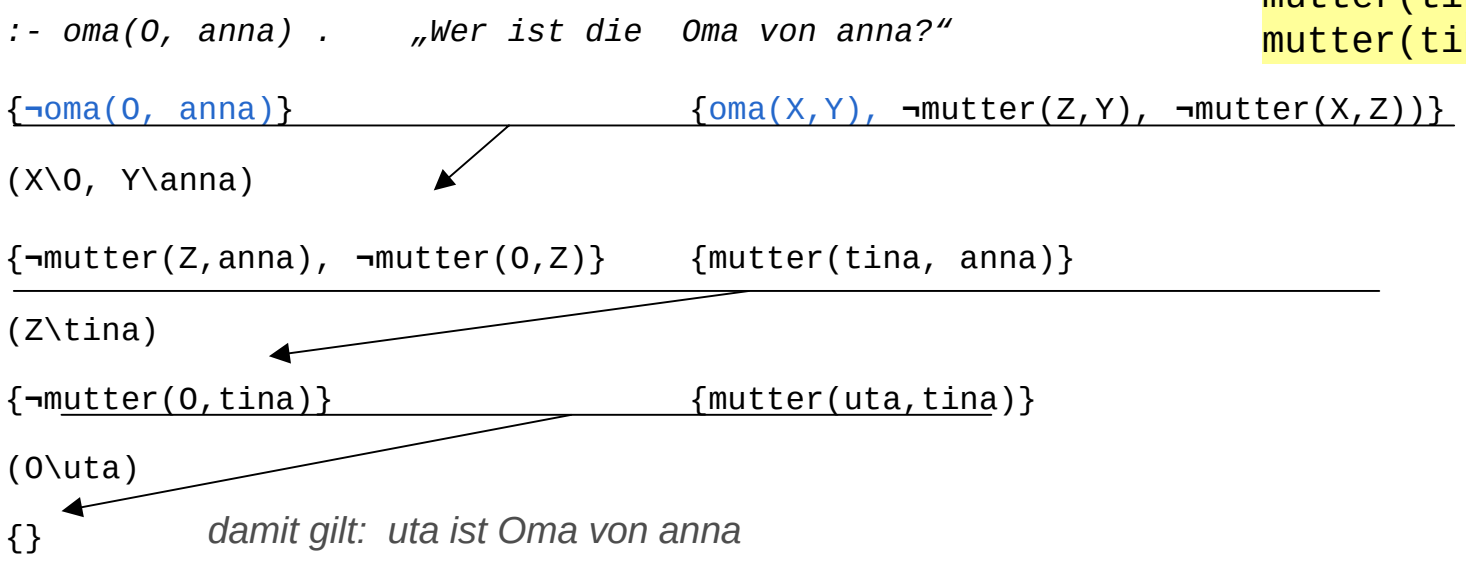
No
7 ?-
```

Beantwortung von Anfragen in Prolog

- Bei der Beantwortung einer Anfrage führt Prolog einen Resolutionsbeweis (SLD-Resolution) durch.
- Falls Anfrage Variablen enthält, wird durch systematisches Ausprobieren nach einer geeigneten Belegung gesucht, mit der unifiziert werden kann.
- Falls mit einer Belegung kein Widerspruch ableitbar ist, erfolgt Backtracking und Prolog wählt eine andere Belegung.

```
mutter(uta, tino).
mutter(uta, tina).
mutter(tina, anna).
mutter(tina, pete).
```

Beispiel:



Beantwortung von Anfragen in Prolog

- Die Bearbeitung einer Anfrage ist abhängig von der Reihenfolge der:
  - Fakten und Regeln im Prolog-Programm
  - Literale in einer Klausel
  - Variablen in einem Literal.

  => Auswirkung auf das Laufzeitverhalten und Reihenfolge der gefundenen Resultate !

Beispiel/Aufgabe: vervollständigen Sie die Oma Regeln

```
oma(X, Y) :-  mutter(X, Z) , mutter(Z, Y) .
```

Disjunktion

- Mit Hornklauseln kann man **keine alternativen Prämissen (Disjunktion)** ausdrücken.
  - **Beispiel:**

    Der darzustellende Sachverhalt sei: "Elternteil ist Vater oder Mutter"

    Darstellung in PL1: $\forall X$ elternteil (X) $\leftarrow$ vater(X) $\vee$ mutter(X)

    aber ist keine Hornklausel

Erlaubt sind in Prolog jedoch Regeln der Form **:- A ; B.** Das Zeichen ";" steht für $\vee$ Interpretation: zunächst wird A bewiesen, schlägt dies fehl, so versucht man B zu beweisen.

```
elternteil (X,Y) :- vater(X,Y);mutter(X,Y).
```

Oder: als zwei Regeln, die dann automatisch durchprobiert werden

```
elternteil(X,Y) :- vater(X,Y).
elternteil(X,Y) :- mutter(X,Y).
```

Rekursion, Suchstrategie

- Benötigt man eine rekursiv definierte Regel, so ist zu beachten, dass Prolog eine Tiefensuche durchführt -> Vorsicht bei Linksrekursion.

Beispiel:

```
vorfahre(X,Y) :- vorfahre(X, Z), elternteil(Z, Y) .
vorfahre(X,Y) :- elternteil(X,Y).
```

Anfrage: *:- vorfahre(X, anna)*

liefert: ERROR: Out of local stack

Modifikation:

```
vorfahre(X,Y) :- elternteil(X,Y).
vorfahre(X,Y) :- elternteil(X,Z), vorfahre(Z, Y)) .
```

Anfrage: *:- vorfahre(X, anna)*

liefert: tina;ulf;uta

Lösung: nur das am weitesten rechts stehende Prädikat darf rekursiv sein

oder

es muss eine zweite Klausel für den Abbruch der Rekursion angegeben werden (vorher).

Gleichheit und Ungleichheit, beliebig

- Es soll bestimmter Wert genommen (unifiziert) werden.
  - Bsp: Vererbung an Nachfolger mit Name „Tina“

```
erbe(X,Y) :- vorfahre(Y,X), X = tina.
```

Unifikation

  - Bsp: Enterben von Tina

```
erbe(X,Y) :- vorfahre(Y,X),X \= tina.
```

- Eine "beliebige" Unifikation: spezielle Variable "_"
  - Beispiel: alle deren Vater bekannt ist

```
kennt_vater(X) :- vater(_,X).
```

Vater zu X in Faktenbasis enthalten

Problem der Eindeutigkeit

- Es werden alle möglichen Unifikationen gefunden. Aus diesem Grund ergeben sich manchmal nicht „logische“ Lösungen

  - Beispiel Geschwister
    ```
    Geschwister(X,Y) :- elternteil(Z,X),elternteil(Z,Y).
    ```
  - Anfrage
    ```
    :-geschwister(X,anna).
    ```
  - Liefert
    ```
    X = pete;
    X = nimo;
    X = anna;
    false
    ```
  - Korrektur
    ```
    geschwister(X,Y) :- elternteil(Z,X),elternteil(Z,Y),X \=Y.
    ```

Problemfall Negation

- Mit Hornklauseln kann man **keine Negation (NOT, NICHT)** ausdrücken.

  Beispiel:

  Der darzustellende Sachverhalt sei:

  "uta ist Mutter aber ulf ist nicht Vater"

  Darstellung in PL1: ¬vater(ulf,tina)

  aber: `mutter(uta,X), ¬vater(ulf,X)` ¬ ist keine Hornklausel

- In Prolog kann man Klauseln mit negierte Prämissen zulassen, wenn man den Inferenzmechanismus um eine geeignete Interpretation des NOT-Operators erweitert.

  D.h. man erlaubt Fakten und Regeln der Form:

  ```
  unehelich(X) :- mutter(uta,X),not(vater(ulf,X)).
  ```

  $p_0(X_0) \leftarrow p_1(X_1), p_2(X_2), ..., p_n(X_n),$ **not**$(p_{n+1}(X_{n+1})), ...,$ **not**$(p_{n+m}(X_{n+m}))$

Negation-as-Failure

- Für Hornklauseln mit Negation gibt es verschiedene Interpretationen.
- Die gängigste ist die ***Negation-as-Failure*** (NAF) Interpretation.
  - NAF setzt die Closed-World-Assumption (CWA) voraus.
  - Umgangssprachlich bedeutet die NAF:

    „*not(G) folgt aus der Wissensbasis WB, wenn der Beweis von G in endlicher Zeit scheitert.*"

    Demnach steht not(G) für „*G ist nicht beweisbar*" und mit der CWA: „*da G nicht aus der WB folgt, muss G falsch sein.*"

Beispiel:

- Wissensbasis mit folgenden Einträgen:

  Regel:
  ```
  loves(john,X) :-  girl(X), not(likes(X,cigarettes)).
  ```

  Fakten:
  ```
  girl(mary).
  girl(susan).
  likes(susan,cigarettes).
  ```

- Anfrage:
  ```
  :- loves(john,Y).
  ```

Übung

- Modellieren in Prolog

Fakten

Diskursbereich

Regeln

```
outermost (X) :- pupet(X),not(contains(_,X))
innermost (X) :- pupet(X),not(contains(X,_))
embeds (X,Y)  :- contains(X,Y).
                 contains(X,Z),embeds (Z,Y).
```

Arithmetik, vordefinierte Prädikate

Arithmetik

- is

  Zuweisung eines **Wertes**: X is 5 + 2
  -> wird Wert 7 zugewiesen. Berechnung kann Variablen enthalten, diese müssen aber bereits einen Wert enthalten.
  Einer Variablen darf nur einmal ein Wert zugewiesen werden.

  "instanziert" werden

- + - * /

  normale Bedeutung

- = \= > >= < =<

  gleich, ungleich, grösser, etc.

- Bsp:

```
X is 5, Y is 8, Z is X + Y
plus(X,Y,Z) :- Z is X + Y.
```

X->5, Y -> 8. Z -> 13

```
plus(3,4,7)
plus(3,4,X)
plus(3,X,7)
```

wahr
X-> 7
nicht erlaubt

Beispiel Fakultät

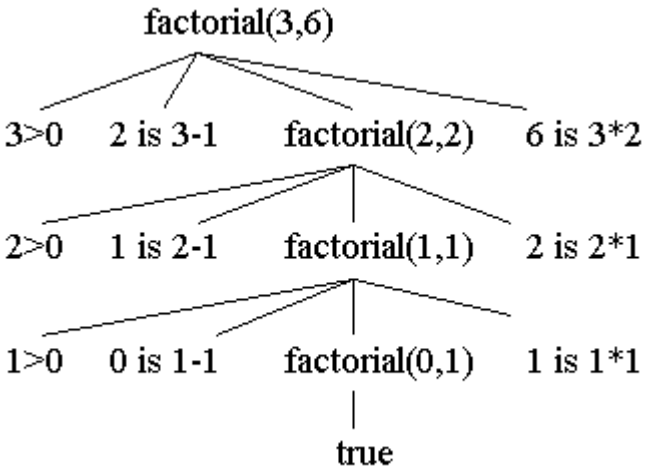
Werte können nur einmal zugewiesen (instanziert) werden
deshalb Fakultät rekursiv berechnen

Was aber nicht funktioniert:

```
factorial(N, R) :-
    N > 0,
    N is N-1,  % immer falsch
    factorial(N, R),
    R is N*R.  % falsch ausser wenn N = 1!
```

Richtig

```
factorial(0,1).
factorial(N,F) :-
   N>0,
   N1 is N-1,
   factorial(N1,F1),
   F is N * F1.
```



http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/2_2.html

Listen

Prolog-Datenstruktur: Liste

- Die Datenstruktur Liste wird in vielen Anwendungen zur Problembeschreibung sowie zur Darstellung einer Lösung benötigt.
- In Prolog stellt man Listen wie folgt dar:
  - Listen werden durch eckige Klammern spezifiziert;, z.B. X = **[** anna,pete,nimo **]**.
  - Listen können als Argumente auftreten, z.B.: **p([**anna,pete,nimo**])**
  - als verkettete Liste der Form **[Head | Tail].** Dabei steht „**|**“ für den sog. Restlistenoperator.

Beispiel: Liste der Kinder als Fakt: `p([anna,pete,nimo]).`

Anfragen:

```
p([X,Y,Z]).
X = anna
Y = pete
Z = nimo

p([X|Y]).
X = anna
Y = [pete, nimo]
```

```
p([X,Y|Z])
X = anna
Y = pete
Z = [nimo]
```

Leere Liste: []

Arbeiten mit Listen

Summierung aller Werte in einer Liste: **`sumlist(L,S)`**

```
sumlist([],S) :- S is 0.
sumlist([X|Y],S) :- sumlist(Y,S1),S is S1 + X.
```

`sumlist([2,4,5,3],S).`

- Ist X Bestandteil von Liste: **`member(X,L)`**

`member(X,[Y|_])  :- X = Y.` wird meist einfach als `member(X,[X|_]).` geschrieben

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

`member(3,[4,5,3]).`

Weitere Listenfunktionen

- Ferner benötigt man Operationen auf Listen, die je nach Prolog-System bereits vordefiniert sind bzw. selbst zu definieren sind.

  Beispiel: **`append(L1, L2, L1undL2).`**

```
append([], L, L).

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

  Beispielaufruf:

```
?- X = [1, 2, 3].

?- append(X, X, Y).

Y = [1, 2, 3, 1, 2, 3]
```

- Erzeugen eines Sets (Menge) mit Werten mit allen möglichen Unifikationen:
  `setof(Template, Goal, Set)`

  Beispiel: alle Kinder von tina

```
setof(X,mutter(tina,X),S)

S = [anna, nimo, pete] ;
```

Elimination von Duplikaten in Lösung

Eingebaute Listenoperationen

School of Engineering

`append(List1, List2, List3)`	fügt Liste1 und Liste2 zu Liste3 zusammen
`member(Element,List)`	Element ist in Liste enthalten
`delete(List1, Element ,List2)*`	Lösche alle übereinstimmenden Elemente aus Liste
`flatten(List1,List2)`	Umformung von geschachtelter Liste in flache
`set(List1, List2)`	Alle Duplikate entfernen
`permutation(List1, List2)`	Erzeuge Permutation von Liste
`reverse(List1, List2)`	Liste umkehren
`last(list,X)`	letztes Element aus Liste
`subtract(List1,List2,List3)`	Lösche Element aus Liste2 die in List1 enthalten sind
`sumlist(List, N)`	Addiere alle Zahlen in Liste
`union(List1,List2, List3)`	Vereinigung zweier Listen
`setof(...)`	Alle Lösungen als Set (ohne Duplikate)
`bagof(...)`	Alle Lösungen als Bag (mit Duplikaten)

*** SWI Prolog spezifisch**

Steuerung der Suche

CUT Operator

- Zur Steuerung der Suche während eines Resolutionsbeweises führt man in Prolog ein spezielles Literal „!“ (sprich CUT, oder Cut-Operator) ein, das immer erfüllt ist und als Seiteneffekt bewirkt, dass bereits getroffene Variablenbindungen (Unifikationen) beibehalten werden.

- CUT beschneidet den Suchbaum (und kann dazu benutzt werden, Backtracking abzubrechen).
- CUT um Laufzeit zu **beschleunigen**

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

member(3,[4,3,5,3,4,5]).
true
```

Abbruch der Rekursion sobald erste Übereinstimmung gefunden

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

CUT um Negation darzustellen

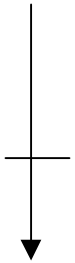
- Anwendung von CUT z.B. um Negation darzustellen, CUT hat Einfluss auf das Resultat

  Beispiel: zu modellieren sei der Sachverhalt:

$$hat_gehalt(X,Y) \wedge \neg\, gehalt(Y, gering) \Rightarrow \neg kredit_wuerdig(X)$$

  Formulierung als Prolog-Regel:

```
kredit_wuerdig(X) :-  hat_gehalt(X,Y),
                      gehalt(Y, gering),
                      !,
                      fail.
```



Abarbeitungsreihenfolge

  Negation: Q :-P,!,fail

```
different(X,Y) :- X=Y,!,fail.
```

Interaktives Arbeiten

Interaktives Arbeiten mit Prolog

- neue Fakten und Regeln können auch mittels ***assert*** *dynamisch bzw interaktiv* zur Faktenbasis hinzugefügt werden (assert entfällt wenn Datei mittels Consult geladen wird)

  Beispiel:

```
assert(vater(ulf, tina)) .
assert(mutter(uta, tino)) .
assert(mutter(uta, tina)) .
assert(mutter(tina, anna)) .
assert(mutter(tina,pete)).
assert(mutter(tina,nimo)).
assert(oma(X,Y):-mutter(X,Z),mutter(Z,Y).
```

- umgekehrt, löscht **retract(...)** *"interaktiv"* Regeln oder Fakten

Ein- und Ausgabe

- `write(X)` schreibe Ausgabe
- `nl` neue Zeile
- Beispiel
  - Ausgabe von allen Kindern
  - ```
    all_kinder(X) :- elternteil(X,Y),
                     write(Y), nl,
                     fail.
    ```
  - fail führt dazu, dass alle (weiteren) möglichen Variablenbindungen ausprobiert werden, i.e. Backtracking ausgelöst wird.
    ```

Ein- und Ausgabe

- read(X) lese ein Atom (Symbol oder Zahl); Eingabe muss mit '.' abgeschlossen werden

Beispiel:

```
input(X,Y) :- write('enter x-value'),nl,read(X), write('enter y-
value'),nl,read(Y).
```

Neue Zeile vor Eingabe zwingend in SWI Prolog!

Prolog Programmierung

- Wissen (= Regeln und Fakten) werden als Hornklauseln notiert
- Tiefensuche beim Beweisen führt dazu, dass die Reihenfolge der Klauseln die Antwort beeinflusst (Extremfall Endlos-Rekursion).
  -> Wissen und Programm-Kontrollfluss sind miteinander verzahnt.
- Prologumgebungen erlauben interaktives Arbeiten, d.h. man stellt im Dialogmodus Anfragen an eine WB und kann diese auch interaktiv modifizieren (mit assert bzw. retract).
- Es gibt auch Prolog Systeme die z.B. in Java eingebunden werden können.
- Kommerzielle Prolog-Systeme bieten meist einen erweiterten Sprachumfang, der deutlich über den Hornkalkül hinaus geht
  - z.B. Konstrukte zur Programmierung von Schleifen
  - Prozess-Verwaltung

Fragen