

Logische Programmierung II

Logisches Paradigma

Prozedural	Objekt-orientierung	Funktional	Logisch	Parallel
<i>Prozeduren</i>	<i>Objekte und OOP</i>	<i>Funktionen</i>	<i>Aussagenlogik</i>	<i>Parallelität</i>
<i>Datentypen und Zeiger</i>	<i>Fünf Konzepte der OOP</i>	<i>Funktionale Konzepte</i>	<i>Resolution und Unifikation</i>	<i>Modelle</i>
<i>Speicher-verwaltung</i>	<i>Anwendung der Konzepte</i>	<i>Streams & Lazy Evaluation</i>	<i>Regelbasiertes Programmieren</i>	<i>Mechanismen</i>
				<i>Analyse</i>
<i>C</i>	<i>Java</i>	<i>Scala</i>	<i>Prolog</i>	<i>Java</i>
Anwendung				

- Grundlegende Syntax von Prolog
- Genereller Ablauf des Programms
 - Unifikation
 - Resolution
 - Probleme mit Rekursion (Tiefensuche in Prolog)
- Was fehlt?
 - Arithmetik
 - Steuerfunktionen

- Prolog unterstützt die Basisoperatoren

Schreibweise	Operation
$X + Y$	Addition
$X - Y$	Subtraktion
$X * Y$	Multiplikation
X / Y	Division
$X \text{ mod } Y$	Rest der ganzzahligen Division

- Arithmetische Ausdrücke werden mit „is“ ausgewertet

```
?- 24 is 21 + 3.
```

```
true.
```

```
?- 2 is 21 + 3.
```

```
false.
```

```
?- X is 7 + 3.
```

```
X = 10
```

```
?- X = 7 + 3.
```

```
X = 7 + 3.
```

```
?- X is Y + 1.
```

```
ERROR: is/2: Arguments are not sufficiently  
instantiated
```

Schreibweise

$X =: = Y$

$X = \backslash = Y$

$X < Y$

$X > Y$

$X = < Y$

$X > = Y$

Operation

numerisch gleich

numerisch ungleich

X kleiner als Y

X größer als Y

X kleiner gleich Y

X größer gleich Y

Schreibweise	Operation
$X = Y$	unifizierbar
$X \backslash = Y$	nicht unifizierbar
$X == Y$	identisch mit
$X \backslash == Y$	nicht identisch mit

- **Achtung: Nicht verwechseln!**

<https://stackoverflow.com/questions/8219555/what-is-the-difference-between-and-in-prolog>

Unterschied: Identifikation und Unifikation

- X wird mit 2 unifiziert – X bekommt also den Wert 2

```
?- X=2.  
X = 2.
```

- es wird geprüft, ob X schon mit Wert 2 unifiziert wurde – es findet keine Unifikation statt

```
?- X==2.  
false.
```

- X wird mit 2 unifiziert und es wird dann geprüft, ob X den Wert 2 hat

```
?- X=2, X==2.  
X = 2.
```

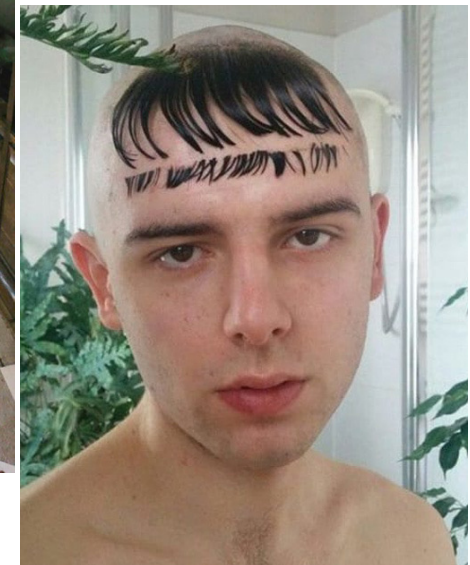

Cut & Fail



hair cut



hair cut fail

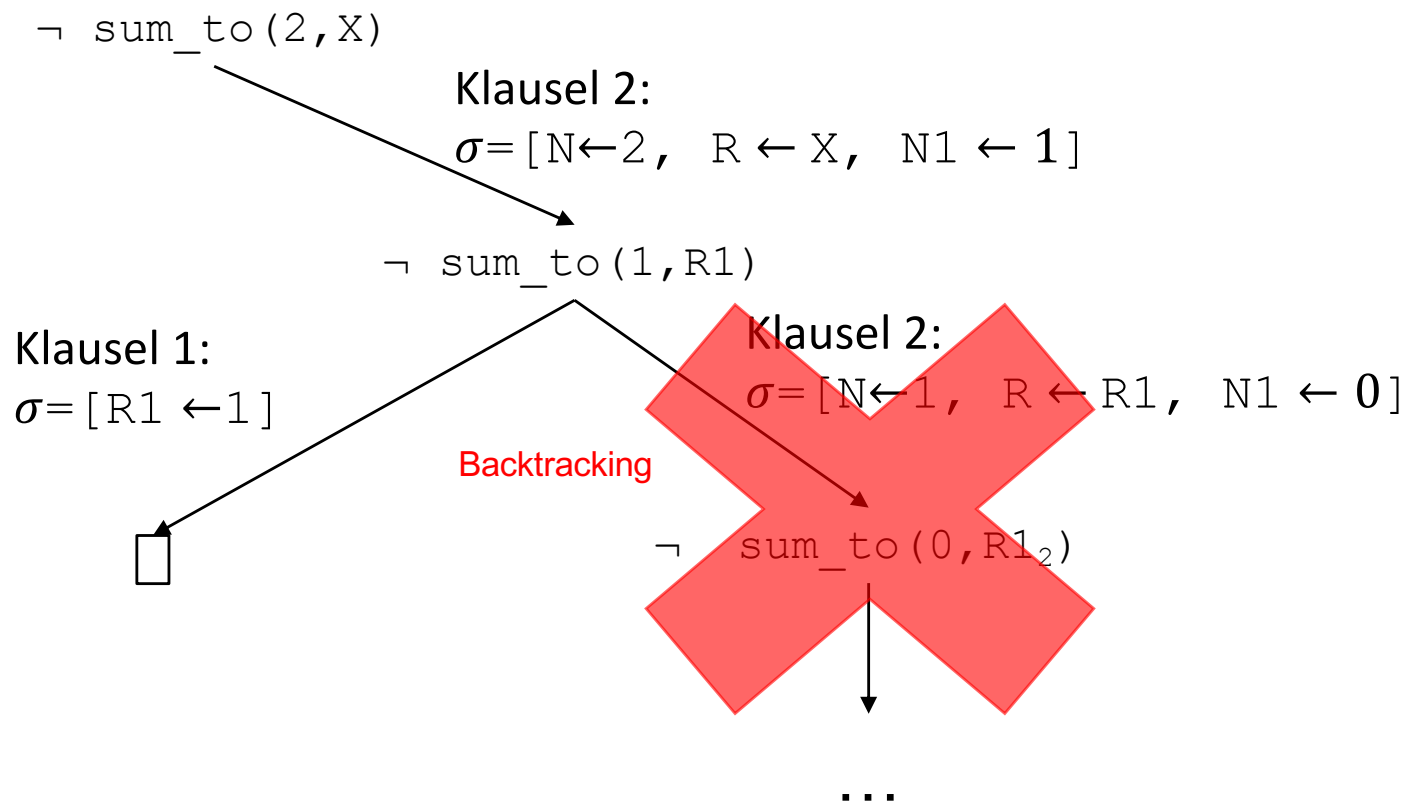


- syntaktisch
 - `foo(...) :- !.`
- semantisch
 - Fixierung aller bisherigen Unifikatoren (bisherige Substitutionen) bis zum Cut
 - dadurch Einschränkung des Backtracking's – Abschneiden von Teilzweigen
 - kein Abbrechen der Regel



`sum_to(N, X)` – X ist die Summe aller Werte von 1 bis N

1. `sum_to(1,1) :- !.`
2. `sum_to(N,R) :- N1 is N-1, sum_to(N1,R1), R is R1+N.`



- ohne Cut:

```
fibNoCut(1, 1).  
fibNoCut(0, 0).  
fibNoCut(N, Value) :-  
    A is N - 1, fibNoCut(A, A1),  
    B is N - 2, fibNoCut(B, B1),  
    Value is A1 + B1.
```

- mit Cut:

```
fib(1, 1) :- !.  
fib(0, 0) :- !.  
fib(N, Value) :-  
    A is N - 1, fib(A, A1),  
    B is N - 2, fib(B, B1),  
    Value is A1 + B1.
```

- **Bestätigung einer Regelauswahl**
 - *Wenn Du soweit gekommen bist, hast Du die richtige Regel für das Ziel ausgewählt.*
- **Fehlschlag eines Ziels ohne Suche nach alternativen Lösungen**
 - *Wenn Du soweit gekommen bist, solltest Du aufhören, dieses Ziel zu verfolgen.*
- **Beenden des Erzeugens alternativer Lösungen durch Backtracking**
 - *Wenn Du soweit gekommen bist, hast Du die Lösung bzw. alle Lösungen gefunden, und es gibt keinen Grund, nach weiteren Alternativen zu suchen.*

- **Bestätigung einer Regelauswahl**

```
sum_to(1,1) :- !.  
sum_to(N,R) :- N1 is N-1, sum_to(N1,R1), R is R1+N.
```

- Beide Regeln treffen bei $N = 1$ zu
 - Cut als Ausdruck der einzig anzuwendenden Regel
 - Gibt es da ein Problem???

- **Alternative:**

```
sum_to(N,1) :- N=<1, !.  
sum_to(N,R) :- N1 is N-1, sum_to(N1,R1), R is R1+N.
```

- **Beenden des Backtrackings**

- `divide(N1, N2, Result)` - ganzzahlige Division

```
divide(LN, N2, 0) :- LN < N2, !.  
divide(N1, N2, Result) :-  
    divide(N1 - N2, N2, RN),  
    Result is RN + 1.
```

- da Ergebnis eindeutig, Cut als Verhinderung des vergeblichen Versuch weitere Lösungen zu finden

- syntaktisch
 - `foo(...) :- fail.`
- semantisch
 - absichtliches Fehlschlagen einer Regel



- **Fehlschlagen lassen eines Ziels ohne Suche nach alternativen Lösungen**

```
enjoys(julia, X) :- big_kahuna_burger(X),!,fail.  
enjoys(julia, X) :- burger(X).  
burger(X) :- big_mac(X).  
burger(X) :- big_kahuna_burger(X).  
burger(X) :- whopper(X).  
big_mac(a).  
big_kahuna_burger(b).  
big_mac(c).  
whopper(d).
```

- Julia mag alle Burger bis auf den Big Kahuna Burger
- Abbruch, wenn kein Ziel mehr gefunden werden kann
- Laufzeitoptimierung

Was ist der Big Kahuna Burger?



source: bbqpit.de

- IF A THEN B ELSE C wird zerlegt in zwei Regeln
 - B :- A und
 - C :- not A

zerlegt

- Cut verhindert „Ausführen des ELSE Zweiges“

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

- Spezielle Syntax A -> B ; C

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

- SLD-Resolution läuft, bis ein Unifikator gefunden wurde, der die Regel erfüllbar macht.
- Was, wenn wir alle Belegungen brauchen?

```
student(jannik) .  
student(yanik) .  
student(iannique) .
```

```
listallstudents :- student(A) ,  
                    write(A) , nl ,  
                    fail.
```

```
listallstudents.
```

- Cut & Fail kann genutzt werden, um Ausnahmen von allgemeinen Regeln zu definieren

```
student(jannik) .  
student(yanik) .  
student(iannique) .
```

```
besteht_klausur(iannique) :- fail.  
besteht_klausur(A) :- student(A) .
```

- Besteht lannique die Klausur?
 - Der Ast mit lannique schlägt fehl, also nein?
 - SLD: schlägt eine Regel fehl: Backtracking!
 - lannique besteht also die Klausur.

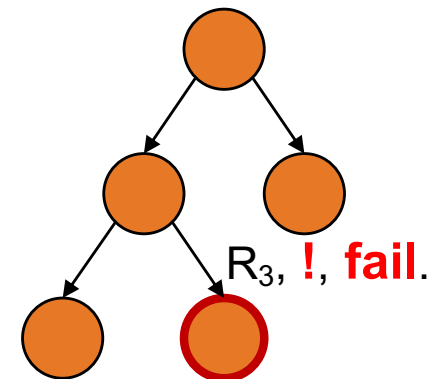
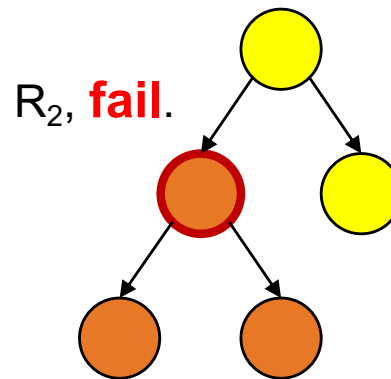
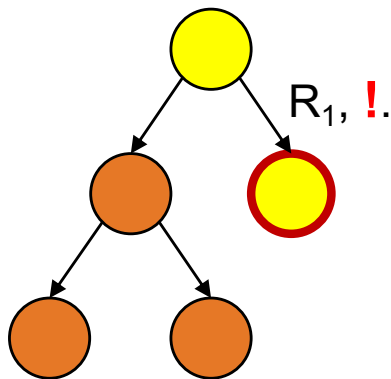
- Cut & Fail kann genutzt werden, um Ausnahmen von allgemeinen Regeln zu definieren


```
student(jannik) .  
student(yanik) .  
student(iannique) .
```


```
besteht_klausur(iannique) :- !, fail.  
besteht_klausur(A) :- student(A) .
```


- Besteht lannique die Klausur?
 - Der Ast mit lannique schlägt fehl, also nein?
 - SLD: schlägt eine Regel fehl: Backtracking!
- Cut & Fail!

- Was sind jetzt die Unterschiede zwischen Cut, Fail und Cut&Fail?



 you are here

 hier könnte sich noch eine möglich Lösung befinden

 hier kann sich keine Lösung befinden



Constraint Logic Programming

$$\begin{array}{rccccccccc} & & & S & & E & & N & & D \\ + & & M & & O & & R & & E \\ \hline M & & O & & N & & E & & Y \end{array}$$

- Gibt es eine linkseindeutige Zuordnung (Injektivität) von Buchstaben/Variablen zu einstelligen Zahlen, sodass die Gleichung erfüllt ist?

```
solve([S,E,N,D,M,O,R,Y]) .
```

Buchstabenrätsel – Generiere und Teste I

- Generiere Zahlen

- Zuordnung Buchstaben zu Zahlen
- vorderste Zahl (Buchstaben S, M) nicht 0

$$\begin{array}{r} S E D \\ + M O R E \\ \hline M O E Y \end{array}$$

```
solve([S,E,N,D,M,O,R,Y]) :-  
digit(S), S>0, digit(M), M>0,  
digit(E), digit(N), digit(D),  
digit(O), digit(R), digit(Y), ... .
```

```
digit(0). digit(1). digit(2). digit(3). digit(4).  
digit(5). digit(6). digit(7). digit(8). digit(9).
```

- Teste injektive Zuordnung
 - unterschiedliche Belegung Buchstaben

	S	E	N	D
+	M	O	R	E
<hr/>				
M	O	N	E	Y

```
solve([S,E,N,D,M,O,R,Y]) :- ... ,  
alldifferent([S,E,N,D,M,O,R,Y]).
```

```
alldifferent([]).  
alldifferent([_,[]]).  
alldifferent([H|T]) :-  
not(member(H,T)), alldifferent(T).
```

- Teste Gleichung

$$\begin{array}{rcccccc} & & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

```
solve([S,E,N,D,M,O,R,Y]) :- ... ,  
add_col(D,E,0,Y,C1),  
add_col(N,R,C1,E,C2),  
add_col(E,O,C2,N,C3),  
add_col(S,M,C3,O,M),  
  
add_col(A,B,C,Res,NC) :- Res is (A+B+C) mod 10,  
                           NC is (A+B+C) // 10.
```

- Ganzzahldivision mit //
- Spalten mit Übertrag (drittes Element in add_col)
- add_col – Summe Res einer Spalten mit zwei Zahlen A, B mit Übertrag NC
- Übergabe des Übertrags über Spalten via C
- M Übertrag der vierten Spalte

Buchstabenrätsel – Generiere und Teste IV

- Vollständige Regel:

$$\begin{array}{r} \\ \\ + \\ \hline M \end{array}$$

```
solve([S,E,N,D,M,O,R,Y]) :-  
  digit(S), S>0, digit(M), M>0,  
  digit(E), digit(N), digit(D),  
  digit(O), digit(R), digit(Y),  
  
  digit(0). digit(1). digit(2). digit(3). digit(4).  
  digit(5). digit(6). digit(7). digit(8). digit(9).  
  
  add_col(D,E,0,Y,C1), add_col(N,R,C1,E,C2),  
  add_col(E,O,C2,N,C3), add_col(S,M,C3,O,M),  
  alldifferent([S,E,N,D,M,O,R,Y]).
```

- Generiere und Teste ineffizient
- **Worst Case:** Bestimmung aller möglichen Tupel und Überprüfung via Backtracking
- *Idee:* vorherige Einschränkung der Möglichkeiten – **Constraint Solver**

- Anwendung clpfd-Bibliothek:

$$\begin{array}{rcccccc} & & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

- Constraint Logic Programming over Finite Domains

```
:- use_module(library(clpfd)).
```

- Formulierung der Constraints

- Wertebereiche:

- Variable: `X in 0..9`
- Listenelemente: `[A,B,C] ins 4..400`

- Relationen:

- `#=` Pendant zu `=`
- `#\=` Pendant zu `\=`
- `#>=` Pendant zu `>=`
- `#<=` Pendant zu `<=`
- `#>` Pendant zu `>`
- `#<` Pendant zu `<`

- arithmetischen Ausdrücken: `+`, `*`, `-`, `//` (Integer-Division), `^`, `min(...)`, `max(...)`, `abs(...)`, ...

- Formulierung der Constraints

```
solve([S,E,N,D,M,O,R,Y]) :- [S,E,N,D,M,O,R,Y] ins 0..9,  
S #\= 0, M #\= 0, ... .
```

- Einschränkung des Wertebereichs soweit wie möglich durch CLP-Operatoren

- Teste injektive Zuordnung und Spalten der Gleichung

$$\begin{array}{rcccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

```
solve([S,E,N,D,M,O,R,Y]) :- ... ,  
all_different([S,E,N,D,M,O,R,Y]),  
D + E #= Y + 10*C1,  
C1 + N + R #= E + 10*C2,  
C2 + E + O #= N + 10*C3,  
C3 + S + M #= O + 10*M.
```

- Vollständige Variante:

$$\begin{array}{rcccccc} & & S & E & N & D & \\ + & M & O & R & E & & \\ \hline M & O & N & E & Y & & \end{array}$$

```
solve([S,E,N,D,M,O,R,Y]) :- [S,E,N,D,M,O,R,Y] ins 0..9,  
S #\= 0, M #\= 0, all_different([S,E,N,D,M,O,R,Y]),  
D + E #= Y + 10*C1,  
C1 + N + R #= E + 10*C2,  
C2 + E + O #= N + 10*C3,  
C3 + S + M #= O + 10*M.
```

- Ergebnis – eingeschränkter Wertebereich durch CLP-Operatoren

```
:- use_module(library(clpfd)).  
?- solve([S,E,N,D,M,O,R,Y]).  
S = 9, M = 1, O = 0, E in 2..8, N in 2..8, D in 2..8, R  
in 2..8, Y in 2..8
```

$$\begin{array}{rccccccccc} & & & S & E & N & D & & & \\ & & & & & & & & & \\ + & & M & O & R & E & & & & \\ \hline M & O & N & E & Y & & & & & \end{array}$$

```
solve([S,E,N,D,M,O,R,Y]) :- [S,E,N,D,M,O,R,Y] ins 0..9,  
S #\= 0, M #\= 0, all_different([S,E,N,D,M,O,R,Y]),  
D + E #= Y + 10*C1,  
C1 + N + R #= E + 10*C2,  
C2 + E + O #= N + 10*C3,  
C3 + S + M #= O + 10*M.
```

- Anzeige konkreter Lösung durch Labeling (systematisches Ausprobieren der Werte im Wertebereich):

```
:- use_module(library(clpfd)).  
?- solve([S,E,N,D,M,O,R,Y]) , label([S,E,N,D,M,O,R,Y]).  
S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2.
```

- Generiere und Teste
 - Generiere alle mögliche Lösungen
 - Teste Erfüllbarkeit der Constraints
- Constraint Logic Programming (CLP)
 - Zähle Constraints auf
 - Einschränkung der Lösungen durch Constraints
 - Wenn endliche Anzahl von Lösungskandidaten, teste diese

- Teilziele - Generiere und Teste:
 - Erfüllung instanziiert Variablen
 - Backtracking, falls unerfüllbar
- Teilziele - CLP:
 - Erfüllung schränkt Wertebereich ein
 - Backtracking, falls Wertebereich leer
 - (noch) nicht anwendbare Constraints werden eingefroren („delayed“)
- **Wichtig:** nicht-leerer Wertebereich beim CLP keine Implikation der Existenz einer Lösung
- Ausschluss möglichst vieler Lösungen durch Constraint-Solver, anschließend normales Backtracking beim Testen

Ist ein Zins mit einer spezifizierten Höhe, einer monatlichen Laufzeit, einer Zinsrate, monatlichen Rückzahlungen und einer Restschuld erfüllbar?

- D – Kredithöhe
- T – Zeit in Monaten
- Z – Zinsrate*
- R – Monatliche Rückzahlrate
- S – Restschuld



mortgage (D, T, Z, R, S)

```
mortgage(D,T,Z,R,S) :- {T = 0, D = S}.  
mortgage(D,T,Z,R,S) :- {T > 0, T1 = T - 1, S = S1  
+ S1*Z - R}, mortgage(D,T1,Z,R,S1).
```

*** Achtung: Die Zinsen fallen monatlich an**

- Constraint Solver `clp[q, r]`
 - `clpq` – Constraints für rationale Zahlen
 - `clpr` – Constraints für reelle Zahlen

```
:- use_module(library(clpq)).  
:- use_module(library(clpr)).
```

- Constraints innerhalb {...}

- Wie hoch ist die monatliche Rate?

```
?- mortgage(10000,12,0.005,R,0) .  
R = 860.664
```

- Wie hoch ist der maximal verfügbare Kredit?

```
?- mortgage(D,12,0.04,12000,0) .  
D = 112620.885
```

- Wie hoch ist die Restschuld?

```
?- mortgage(180000,24,0.04,9000,S) .  
S = 109651.312
```

- Wie lang ist die Kreditlaufzeit?

```
?- mortgage(148000,T,0.04,12000,S),S<0 .  
T = 18.0,  
S = -7924.110337536535
```

- Wie hoch ist der Tilgungsbetrag im ersten Monat?

```
?- mortgage(148000,1,0.08,24000,148000-CP) .  
CP = 12159.999
```



Anwendungsbeispiele Prolog

- PrologBeans

- Einbinden von Prolog in Java

- Backend über Prolog

- z.B. für Webseiten

- Frontend (Layout) über Java Application Server

- Backend (Logik, Datenbank) über Prolog

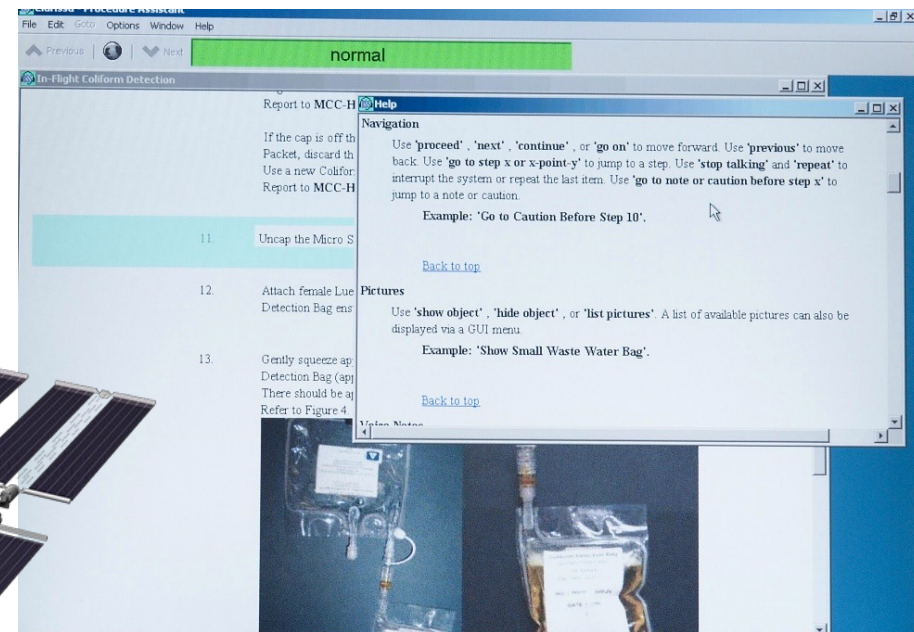
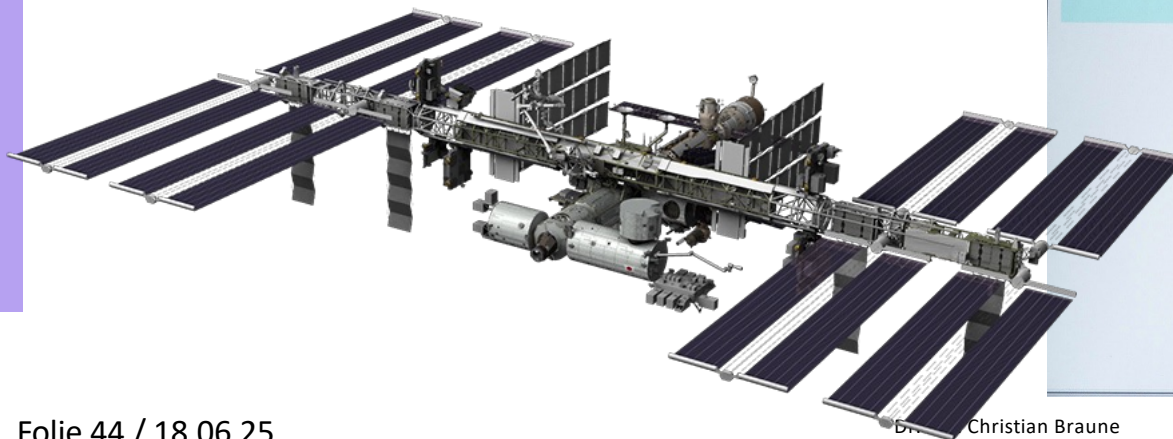
- Vorteile

- Einfacher und extrem kurzer Code für die Abbildung der Businessregeln

https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/lib_002dprologbeans.html#lib_002dprologbeans

- Clarissa Procedure Guidance

- sprachbasiertes System zur Durchführung von Prozeduren
- Schritte der Prozedur werden vorgelesen
- Astronaut kann per Sprache vor und zurück gehen, Bilder anzeigen lassen und Notizen machen
- Clarissa entscheidet selbstständig, welche Pfade in der Prozedur genommen werden müssen, je nach Ergebnis des letzten Schritts
- Seit 2005 auf der IIS im Einsatz



Wie funktioniert denn nun Watson?



***Er ist der Antagonist in Stevenson's
Schatzinsel.***

Wie funktioniert IBM Watson?

Natural Language Processing With Prolog in the IBM *Watson* System

Adam Lally
IBM Thomas J. Watson Research Center

Paul Fodor
Stony Brook University

24 May 2011

On February 14-16, 2011, the IBM Watson question answering system won the Jeopardy! Man vs. Machine Challenge by defeating two former grand champions, Ken Jennings and Brad Rutter. To compete successfully at Jeopardy!, Watson had to answer complex natural language questions over an extremely broad domain of knowledge.

<https://www.semanticscholar.org/paper/Natural-Language-Processing-With-Prolog-in-the-IBM-Lally-Fodor/c0ddc1e420ab0b2477865b160130335aad41dcc6>

Building Watson: An Overview of the DeepQA Project

David Ferrucci, Eric Brown, Jennifer Chu-Carroll,
James Fan, David Gondek, Aditya A. Kalyanpur,
Adam Lally, J. William Murdock, Eric Nyberg, John Prager,
Nico Schlaefer, and Chris Welty

■ IBM Research undertook a challenge to build a computer system that could compete at the human champion level in real time on the American TV quiz show, Jeopardy. The extent

The goals of IBM Research are to advance computer science by exploring new ways for computer technology to affect science, business, and society. Roughly three years ago, IBM Research was looking for a major research challenge to rival

<https://www.aaai.org/ojs/index.php/aimagazine/article/view/2303>

Schritt 0: Frage parsen und in logischer Struktur darstellen

Syntax:

Subjekt – Prädikat – Objekt

Objekt = Atomobjekt oder Attribut&&Objekt oder ...

Attribut = Adjektiv oder Possesivadjektiv um oder ...

Für Fragen (Jeopardy): Subjekt ist gesucht

Er ist der Antagonist in Stevenson's Schatzinsel.

- Schritt 1: Logische Strukturen erstellen
 - X is the answer.
 - antagonist(X).
 - antagonist_of(X, Stevenson's Treasure Island).
 - modifies_possesive(Stevenson, Treasure Island).
 - modifies(Treasure, Island).

Er ist der Antagonist in Stevenson's Schatzinsel.

- Schritt 2: Semantische Annahmen treffen (Heuristik, Lernen)
 - island(Treasure Island)
 - location(Treasure Island)
 - resort(Treasure Island)
 - book(Treasure Island)
 - movie(Treasure Island)
 - person(Stevenson)
 - organization(Stevenson)
 - company(Stevenson)
 - author(Stevenson)
 - director(Stevenson)
 - person(antagonist)

Er ist der Antagonist in Stevenson's Schatzinsel.

- Schritt 3: Anfragen an die Datenbanksysteme basierend auf den Annahmen aus Schritt 2
- Schritt 4: Antworten aus den Ergebnissen aus Schritt 3 ermitteln
 - Eine davon sollte *Long-John Silver* sein

- Schritt 5: Für jede Antwort aus der Datenbank, führe Schritt 2 erneut aus, um Beweise oder Ablehnungsgründe für diese Antwort zu finden
 - Long-John Silver, the main character in Treasure Island.....
 - Treasure Island, by Stevenson was a great book.
 - One of the greatest antagonists of all time was Long-John Silver
 - Richard Lewis Stevenson's book, Treasure Island features many great characters, the greatest of which was Long-John Silver.

Grundidee: unvollständige Wissenbasis bzw. nicht-eindeutige Frage, Gewichtung der verschiedenen Lösungen

➔ je mehr Regeln zu einer Lösung führen, um so wahrscheinlicher die Lösung

- Schritt 6: Generiere neues Wissen, Beweise dieses Wissen und Bewerte die Antworten neu
 - Stevenson = Richard Lewis Stevenson
 - “by Stevenson” → Stevenson’s
 - main character → antagonist

- Schritt 7: Kombiniere alle Beweise und berechne Konfidenz
 - Auswertung der Antworten bzgl. ihrer Wahrscheinlichkeiten
 - Ausgabe der wahrscheinlichsten Antwort

- Wissensbasis und Regeln
 - typischerweise eingeschränkte Wissensbasis
 - Regeln formal definite Hornklauseln
- Erstellung von Wissensbasis und Regeln
 - Generiere und Teste
 - Constraint Logic Programming
- Anfragen an Wissensbasis möglich
 - „Generierung“ von neuem Wissen basierend auf Wissensbasis und Regeln
 - Auswertung via SLD-Resolution
- diverse Anwendungsbereiche
 - Expertensysteme, Data Mining, Jeopardy gewinnen...