

Künstliche Intelligenz

Sebastian Iwanowski
FH Wedel

Kap. 2:
KI-Logik

2.1: Crashkurs PROLOG

Literatur zu Prolog

Lehrbücher:

Ivan Bratko: *PROLOG, Programming for Artificial Intelligence*,
2nd Edition, Wiley 1990, ISBN 0-201-41606-9
3rd Edition, Wiley 2000, ISBN 0-201-40375-7
4th Edition, Wiley 20??, ISBN 0-321-41746-1

Peter Bothner / Wolf-Michael Kähler: *Programmieren in PROLOG*,
Eine umfassende praxisgerechte Einführung,
Vieweg 1991, ISBN 3-528-05158-2

P. Blackburn, J. Bos, K. Striegnitz: *Learn Prolog Now!*,
Texts in Computing Vol. 7, King's College Publications. 2006, ISBN 1-904987-17-6.
Companion website mit On-line-Version: www.learnprolognow.org

Seminararbeit:

Max Rohde: *Eignung logischer Programmiersprachen für Spiele-KI am Beispiel Prolog*,
FH Wedel, Iwanowski, SS 2007, Informatik-Seminar zur Spiele-KI

↳ gibt auch einen Überblick über Prolog und enthält weiterführende Literaturliste

Bausteine von PROLOG

Elementarbausteine:

- **Zahlen**
Unterschieden wird zwischen ganzen und gebrochen rationalen Zahlen.
- **Atome**
Name, dessen erstes Zeichen Kleinbuchstabe ist.
- **Variable**
Name, dessen erstes Zeichen Großbuchstabe ist. Ausnahme: `_`
- **Listen**
[] oder [Term | Liste]
Kurzschreibweise: [1,2,3,4] für [1 | [2 | [3 | [4 | []]]]]
- **Terme**
Zahlen, Atome, Variable, Listen oder Atom(Term) oder Atom(Term,Term) oder ...
- **Prädikate**
Terme der Form Atom, Atom(Term) oder Atom(Term,Term) oder ...
2 Prädikate gelten als gleich, wenn sie mit demselben Atom benannt sind und dieselbe Anzahl von Parametern haben.

Bausteine von PROLOG

Logische Operatoren zwischen Prädikaten:

- **Konjunktion**

a , b entspricht: $a \wedge b$

- **Implikation**

a :- b entspricht: $b \rightarrow a$

- **Äquivalenz**

a = b entspricht: $b \leftrightarrow a$

- **Nichtäquivalenz**

a \= b entspricht: $b \nleftrightarrow a$

- **versionsspezifische Operatoren**

not, member, length, ...

Bausteine von PROLOG

Arithmetische Operatoren

- **+, -, *, /, div, mod**

Arithmetische Ausdrücke werden in Infix-Notation gebildet.

Auswertung arithmetischer Operationen

- **nicht automatisch!**
- **durch Zuweisung an Variable**

Varname **is** Arithmetischer Ausdruck
weist der Variable Varname das Ergebnis des arithmetischen Ausdrucks zu.

- **durch logische Operatoren mit Auswertungsfunktion**

<, =<, > >=, :=, =\= wertet arithmetische Ausdrücke auf beiden Seiten aus.
(in manchen Implementierungen nur auf einer Seite)

Bausteine von PROLOG

Wissen in Form von Klauseln

- **Fakten**

Prädikat.

Derartige Prädikate werden in der Wissensbasis als wahr vorausgesetzt.

- **Regeln**

Prädikat :- Konjunktion von Prädikaten.

Derartige Terme werden in der Wissensbasis als wahr vorausgesetzt, wenn die rechte Seite als wahr vorausgesetzt werden muss.

Es kann für dasselbe Prädikat als Konklusion mehrere Regeln geben.

- **Fragen**

?- Konjunktion von Prädikaten.

Prolog versucht, eine Frage aus den bekannten Fakten und Regeln herzuleiten. Falls das gelingt, kommt als Antwort `yes` mit der dafür notwendigen Unifikation für die Variablen, anderenfalls `no`.

Funktionsweise eines PROLOG-Interpreters

PROLOG ist wissensbasiert:

- **Wissensbasis**

Fakten und Regeln, dynamisch erweiterbar

- **Inferenzmaschine**

Automatische Herleitung neuer Fakten und Regeln mit Resolution und Unifikation

- **Dialogkomponente**

Eingabe: Fragen

Ausgabe: yes / no, Angabe der Unifikation im Erfolgsfall, Write als „Seiteneffekt“

Yes: Das Prädikat der Frage folgt aus der Wissensbasis.

No: Das Prädikat der Frage folgt nicht aus der Wissensbasis.

No impliziert nicht, dass das Prädikat als falsch abgeleitet werden kann.

Funktionsweise eines PROLOG-Interpreters

Arbeitsweise der Inferenzmaschine:

- **Zerlegung eines Ziels in Unterziele**

Erstes Ziel ist die Frage.

Versuch, das Ziel durch Unifikation mit Prädikaten aus der Wissensbasis zu erreichen.

Auswertung von Regeln führt zu Unterzielen.

- **Auswertungsreihenfolge**

Alle Daten der Wissensbasis werden **von oben nach unten** ausgewertet.

Konjunktionen in Regelvoraussetzungen werden **von links nach rechts** ausgewertet.

Die Auswertungsreihenfolge wird *nicht* getrennt nach Fakten und Regeln vorgenommen.

- **Instanziierung von Variablen**

Variablen werden nur zum Zweck der Unifikation mit Werten instanziiert.

Die Instanziierung wird nach Misserfolg des gegenwärtigen Suchzweigs wieder aufgehoben.

- **Backtracking**

Nach dem Misserfolg einer gegenwärtigen Instanziierung wird eine neue Instanziierung versucht.

Tiefes Backtracking: Anderer Wert zur Erfüllung derselben Klausel.

Seichtes Backtracking: Anderer Wert zur Erfüllung einer anderen Klausel für dasselbe Prädikat.

Einfache Beispiele

- **Prädikatenwelt a la GTI:**

Wissensbasis:

vater(sven, georg).
bruder(holger, anna).
verheiratet(sven, anna).

maennlich(X) :- vater(X,Y).
maennlich(X) :- bruder(X,Y).

onkel(X,Y) :- vater(Z,Y), bruder(X,Z).
onkel(X,Y) :- mutter(Z,Y), bruder(X,Z).
mutter(X,Y) :- vater(Z,Y), verheiratet(X,Z).
weiblich(X) :- verheiratet(X,Z), maennlich(Z).
verheiratet(X,Y) :- verheiratet(Y,X).

Fragen:

?-weiblich(X).
?-maennlich(X).
?-onkel(holger,X).
?-verheiratet(X,Y).

Deklarative Lösung ohne die Probleme mit symmetrischen Prädikaten: XSB

<http://xsb.sourceforge.net/>

In ISO-Prolog funktioniert das nicht!

Besser:

verh(X,Y) :- verheiratet(X,Y).
verh(X,Y) :- verheiratet(Y,X).

?- verh(X,Y).

Einfache Beispiele

- **Größter gemeinsamer Teiler:**

Wissensbasis:

$\text{ggt}(X, X, X).$

$\text{ggt}(X, Y, D) :- X < Y, Y1 \text{ is } Y - X, \text{ggt}(X, Y1, D).$

$\text{ggt}(X, Y, D) :- Y < X, \text{ggt}(Y, X, D).$

Frage:

?-ggt(234,330,Ergebnis).

- **Listenlänge:**

Wissensbasis:

$\text{length}([], 0).$

$\text{length}([\text{Head} | \text{Tail}], N) :- \text{length}(\text{Tail}, N_{\text{minus1}}), N \text{ is } N_{\text{minus1}} + 1.$

Frage:

?-length([1,3,5,8],Ergebnis).

Einflussnahme auf das Backtracking

- **Prädikat fail fürs erschöpfende Backtracking**

Anwendung:

Zum Errechnen mehrerer Lösungen, häufig in Verbindung mit write-Befehlen

Bsp.:

?-maennlich(X), write(X), nl, fail.

- **Cut-Operator ! zum Abschneiden von Suchzweigen**

Anwendung:

Effizienzgewinn (**grünes Cut**), Korrektheitsbeeinflussung (**rotes Cut**)

Bsp.: Finden „aller“ Lösungen zur Gewinnung des Maximums aus 2 Zahlen.

$\text{max}(X, Y, Y) \text{ :- } X \leq Y, !.$

$\text{max}(X, Y, X) \text{ :- } Y \leq X.$

$\text{max}(X, Y, Y) \text{ :- } X \leq Y, !.$

$\text{max}(X, Y, X).$

Einflussnahme auf das Backtracking

Bsp. für grünes Cut: Alle Lösungen für Fibonaccizahlen:

```
fib(0,0).
fib(1,1).
fibRec(N,Res):- fib(N,Res),!.
fibRec(N,Res) :- N > 1,
    N1 is N-1,
    N2 is N-2,
    fibRec(N1,R1),
    fibRec(N2,R2),
    Res is R1+R2,
    asserta(fib(N,Res)).
```

siehe auch:

<http://cs.union.edu/~striegnk/learn-prolog-now/slides/kris/>

```
?-fib(500,X).
?-fibRec(1000,X).
?-fib(500,X).
?-retract(fib(X,Y)), fail.
```

Anmerkung: `asserta`(Prädikat) fügt ein Prädikat der Faktenbasis am Anfang hinzu,
`retract` (Prädikat) entfernt das Prädikat wieder.

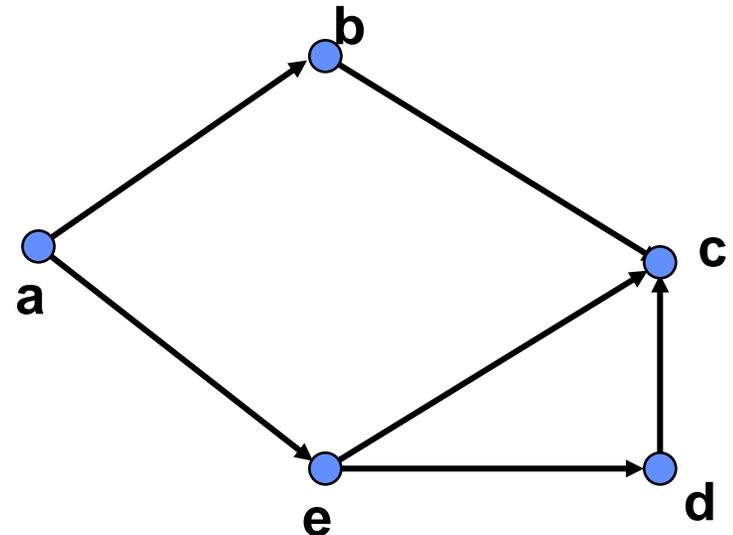
Bsp. für rotes Cut: nicht(X):

```
nicht(X) :- X, !, fail.
nicht(X).
```

Bsp. aus der Graphentheorie

Berechnung aller erreichbaren Knoten von einer Quelle a:

```
kante(a,b).  
kante(a,e).  
kante(b,c).  
kante(e,d).  
kante(e,c).  
kante(d,c).
```



```
verbunden(X,X,[X]).  
verbunden(X,Y,[X|Weg]) :- kante(X,Z), verbunden(Z,Y,Weg).
```

```
% Im Folgenden werden alle möglichen Wege von a aus in Output geschrieben:  
?-verbunden(a,X,Weg), write(Weg), nl, fail.
```

```
% Im Folgenden werden in Liste die von a aus erreichbaren Knoten eingesammelt.  
% Jeder Knoten wird so häufig gezählt, wie er in verschiedenen Wegen erreichbar ist:  
?-findall(Y,verbunden(a,Y,Weg),Liste).
```

Bsp.: 8-Damen-Problem

- 1. Verfahren (Bratko): Greedy-Strategie

Wissensbasis:

queens1([]).

```
queens1([X/Y | Others]) :-  
  queens1(Others),  
  member(Y,[1,2,3,4,5,6,7,8]),  
  conflictFree(X/Y,Others).
```

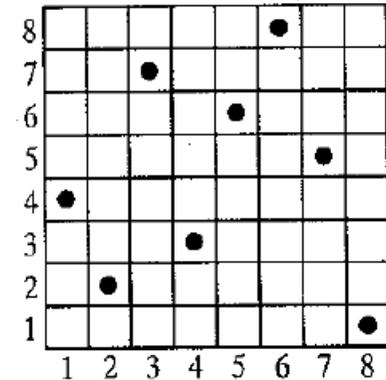
```
conflictFree(_,[]).
```

```
conflictFree(X/Y, [HeadX/HeadY | Others]) :-  
  Y \= HeadY,  
  DiffY is HeadY - Y,  
  DiffY \= HeadX - X,  
  DiffY \= X - HeadX,  
  conflictFree(X / Y,Others).
```

```
template([1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8]).
```

Frage:

?-template(S), queens1(S).



Nicht: $\text{DiffY} = \text{HeadY} - Y$

Nicht: $\text{HeadY} - Y = \text{HeadX} - X$

Bsp.: 8-Damen-Problem

- 3. Verfahren (Bratko): Effizientere Strategie

Wissensbasis:

queens3(YList) :-

```
sol(YList, [1,2,3,4,5,6,7,8],  
     [1,2,3,4,5,6,7,8],  
     [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7],  
     [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]).
```

sol([],[], DomainY, DomainU, DomainV).

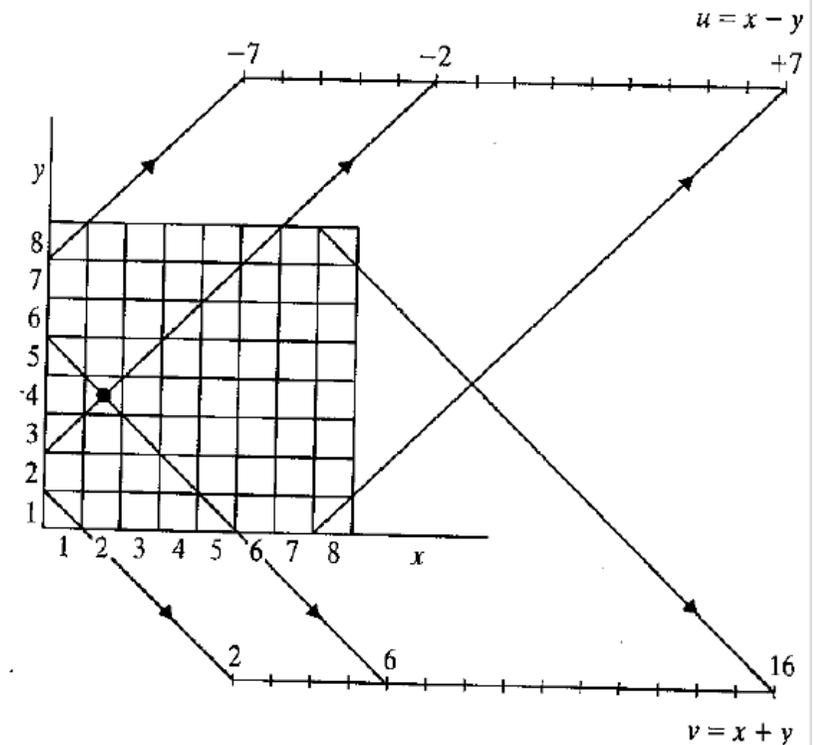
```
sol([Y | YTail], [X | XTail], DomainY, DomainU, DomainV) :-  
  del(Y,DomainY,ReducedDomainY),  
  U is X - Y,  
  del(U,DomainU,ReducedDomainU),  
  V is X + Y,  
  del(V,DomainV,ReducedDomainV),  
  sol(YTail, XTail, ReducedDomainY, ReducedDomainU,  
      ReducedDomainV).
```

del(Item, [Item|List], List).

```
del(Item, [First|Tail],[First|ResultTail]) :-  
  del(Item,Tail,ResultTail).
```

Frage:

?-queens3(YList).



Bsp.: 8-Damen-Problem

- 2. Verfahren (Bratko): Sehr uneffiziente Strategie

Wissensbasis:

```
queens2(YList) :-  
  permutation([1,2,3,4,5,6,7,8], YList),  
  admissible(YList).
```

```
permutation([], []).  
permutation([First|Tail], ResultList) :-  
  permutation(Tail, ResultTail),  
  del(First, ResultList, ResultTail). %del siehe 3. Verfahren
```

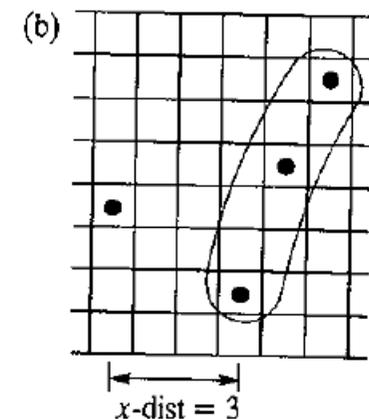
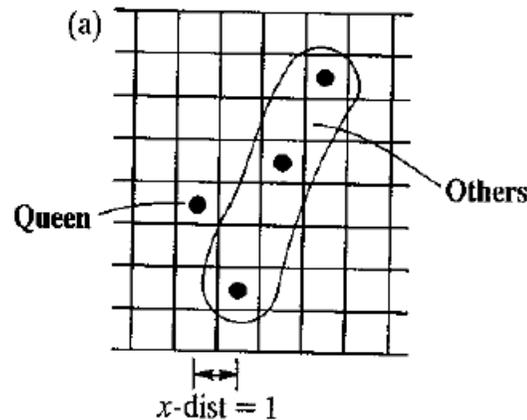
```
admissible([]).  
admissible([Y1|Others]) :-  
  admissible(Others),  
  conflictFree(Y1, Others, 1).
```

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

```
conflictFree(Y, [Y1|YTail], XDiff) :-  
  YDiff is Y1 - Y,  
  YDiff =\= XDiff,  
  YDiff =\= -XDiff,  
  XDiff1 is XDiff + 1,  
  conflictFree(Y, YTail, XDiff1).
```

Frage:

?-queens2(YList).



Bsp.: 8-Damen-Problem

- 4. Verfahren(Max Rohde): Rein deklarativ

Wissensbasis:

zahl(1). zahl(2). zahl(3). zahl(4). zahl(5). zahl(6). zahl(7). zahl(8).

Frage:

?-damen4(8,[],Ergebnis).

eins_weniger(Zahl, ZahlMinusEins):-

zahl(Zahl),

(zahl(ZahlMinusEins);ZahlMinusEins=0), % ; entspricht Disjunktion

(Zahl - 1) =:= ZahlMinusEins.

sichere_position(_, []).

sichere_position(p(X1, Y1), [p(X2, Y2)|R]):-

zahl(X1), zahl(Y1), zahl(X2), zahl(Y2),

X1=\=X2, Y1=\=Y2, % keine gleichen Zeilen/Spalten

Dx is X1-X2, Dy is Y1-Y2, % keine gleiche Diagonale

Dx =\= Dy, Dx =\= (-Dy),

sichere_position(p(X1, Y1), R).

sichere_stellung([]).

sichere_stellung([F|R]):-sichere_position(F, R), sichere_stellung(R).

damen4(0, Ergebnis, Ergebnis).

damen4(N, Start, Ergebnis):- zahl(N), zahl(Y),

sichere_stellung([p(N, Y)|Start]),

eins_weniger(N, NMinusEins),

damen4(NMinusEins, [p(N, Y)|Start], Ergebnis).

Bsp.: 8-Damen-Problem

- Gleiches Verfahren in Java (Max Rohde):

```
/**
 * Adds (or removes) a queen to the given coordinates.
 * If add is true, it adds a queen. If it is false, it removes a queen.
 */
private void addQueen(int x, int y, boolean add) {
    int i;
    gr = this.getGraphics();
    if (add)
        grid[x][y] = 2;
    else
        grid[x][y] = 0;
    for (i=0;i<8;i++) {
        if (i!=y) changeCell(x,i,add);
        if (i!=x) changeCell(i,y,add);
    }
    for (i=1; legal(x+i,y+i); i++)
        changeCell(x+i,y+i,add);
    for (i=1; legal(x+i,y-i); i++)
        changeCell(x+i,y-i,add);
    for (i=1; legal(x-i,y+i); i++)
        changeCell(x-i,y+i,add);
    for (i=1; legal(x-i,y-i); i++)
        changeCell(x-i,y-i,add);

    drawCell(x,y);
    if (!add)
        for (i=0;i<8;i++)
            for (int j=0;j<8;j++)
                if (grid[i][j] == 2)
                    addQueen(i,j,true);
}
```

```
/**
 * Add or remove a queen to the given cell
 */
private void changeCell(int x, int y, boolean add) {
    if (add && grid[x][y] == 0)
        grid[x][y] = 1;
    else if (!add && grid[x][y] == 1)
        grid[x][y] = 0;
    drawCell(x,y);
}
```

```
/**
 * Returns true if the coordinates are a legal board position
 */
private boolean legal(int x, int y) {
    return (x >= 0 && x < 8 && y >= 0 && y < 8);
}
```

```
/**
 * Solve the 8-queens puzzle with a very simple depth-first search.
 */
public boolean solve(int y) {
    int i,j;
    boolean r = false;

    for (i=0;i<8;i++) {
        // for each row, try placing a queen
        if (grid[i][y] == 0) {
            addQueen(i,y,true);
            if (y == 7) {
                return true; //we have placed all queens successfully
            } else {
                if (solve(y+1)) {
                    return true; // we solved it, return
                } else {
                    addQueen(i,y,false); // remove the queen
                }
            }
        }
    }
    // unable to solve down this branch -- retreat!
    return false;
}
```

Nutzen von Prolog

Didaktischer Nutzen:

- **Schulung im Umgang mit formaler Logik**
- **Übung im rekursiven Formulieren von Problemen (und Algorithmen)**

Praktischer Nutzen:

- **gut fürs schnelle Ausprobieren (rapid prototyping)**
- **hauptsächlich geeignet für Probleme, für die es keine anderen Algorithmen gibt als systematisches Ausprobieren.**
- **geeignet zur sukzessiven oder systematischen Ausgabe aller zulässigen Lösungen eines Problems.**

Grenzen:

- **Mehr Spielwiese als kommerziell nutzbares Werkzeug, zu fern von den meisten praktischen Problemstellungen**
- **vollkommen unbrauchbar, wenn es um Effizienz der Lösungsgewinnung geht.**