

Logikprogrammierung

Historie:

- Philosophie / Mathematik
- Künstliche Intelligenz

Praktische Programmiersprache:

Prolog = Programmierung in Logik (1974)

Grundidee:

Ein Programm ist eine Menge von Fakten und Regeln.

Die Anwendung eines Programms besteht in der Beantwortung von Fragen.

Antworten sind entweder nur Ja/Nein oder bestehen darin, dass Platzhalter (Variable) der Frage mit einem möglichen Inhalt belegt werden.

Lernziele

Grundlagenwissen / Begriffe:

- Prädikatenlogik, Unifikation, Resolution

Paradigmen:

- deklarativer Programmierstil
- Ablauf als Lösungssuche

Programmiertechniken:

- Rekursion
- Endrekursion
- Symbolische Ausdrücke
- Listenoperationen
- Backtracking

Prolog ist aus der Prädikatenlogik abgeleitet.

Aussagenlogik enthält Aussagen und deren Verknüpfungen.

In der Prädikatenlogik erster Ordnung, können die Aussagen von formalen Ausdrücken abhängen und Variablen enthalten. Es gibt universell (all-) und existenziell quantifizierte Variable. Die Aussagenfunktionen heißen Prädikat.

In der Prädikatenlogik zweiter Ordnung können die Prädikate selbst durch Variable repräsentiert werden.

z.B. vollständige Induktion:

$$\forall E[(E(1) \& \forall n (E(n) \Rightarrow E(n'))) \Rightarrow \forall n E(n)]$$

Es gibt kein Kalkül (formales System) der Prädikatenlogik 2. Ordnung, das gleichzeitig korrekt und vollständig ist (Gödelscher Satz). Eine vollständig formale Begründung der Mathematik ist unmöglich.

Aussagenlogik und Prädikatenlogik 1. Ordnung können dagegen vollständig formalisiert werden.

Lexikalische Elemente von Prolog

Variable: Bezeichner, die mit einem Großbuchstaben oder `_` anfangen

`X, X1, _X, _alfa, _`

Atome: alle anderen Bezeichner und Bezeichner die in `' '` eingeschlossen sind.

`vater, mutter, 'Vater'`

Zahlen: `1, 1.5, 1e-2`

Kommentare `/* ... */` und `%` für einen Zeilenkommentar

Sonderzeichen:

- `:-` trennt linke und rechte Seite einer Regel
- `,` Trennzeichen bei Aufzählungen.
- `.` beendet eine Regel
- `?-` leitet eine Frage ein.
- `()` Klammern
- `[]` Listenklammern
- `|` trennt das erste von den restlichen Listenelementen
- `!` Cut
- `...` Arithmetik etc.

Strukturen

Prolog-Ausdrücke (structure) bestehen aus einem Funktor (Atom) optional gefolgt von einer eingeklammerten und durch Komma getrennten Liste von Argumenten. Die Anzahl der Argumente bestimmt die Stelligkeit (arity) des Funktors.

Beispiele:

```
hans
es_regnet
es_regnet(gummersbach)
besitzt(hans, buch(goethe, faust))
anschrift(deutschland, koeln, chlodwigplatz)
```

arity: hans/0, es_regnet/0, es_regnet/1, anschrift/3

Einige Ausdrücke (insbes. mit Sonderzeichen) werden in Operatorschreibweise geschrieben: $+(4, 3)$ ist gleichbedeutend mit $4 + 3$

Operatoren / Präzedenz und syntaktische Hilfsmittel

Prolog verfügt über ein bereits vordefiniertes aber erweiterbares System von Regeln wie Ausdrücke mit Operatoren zu verstehen sind.

Beispiele:

$3 + 4 * 5$ wird gelesen als $+(3, *(4, 5))$
 $[1,2,3]$ wird gelesen als $.(1, .(2, .(3, [])))$
 $[1 | [2, 3]]$ wird gelesen als $.(1, [2, 3])$

Zum dem Arithmetik-Beispiel muss gesagt werden, dass Prolog von alleine nicht rechnet. Damit ermöglicht es formale Systeme für symbolische Formeln zu entwickeln (Zum Rechnen nutzt man vordefinierte Funktionen).

Die beiden anderen Beispiele zeigen schon mal die Listennotation von Prolog. Sie entspricht dem Modell der einfach verketteten Liste, d.h. man kann sich die $.$ -Struktur als Listenknoten $knoten(Wert, Restliste)$ vorstellen. Wir werden dies noch besprechen und auch in der funktionalen Programmierung wiederfinden.

Literale / Prädikate

Ein Literal ist eine atomare Aussage, die möglicherweise negiert ist (negatives Literal, sonst positives Literal).

Ein Prädikat bezeichnet die mit einem Literal verbundene Aussage. Dabei kann das Prädikat durch mehrere Aussagen präzisiert werden.

Beispiele (in der Prädikatenlogik):

teiler(4, 12), \neg **teiler**(4, 10)

$\forall x \forall y$ elternteil(x, y) \Rightarrow **vorfahr**(x, y)

$\forall x, y, z$ elterternteil(x, y) & vorfahr(y, z) \Rightarrow **vorfahr**(x, z)

Beispiele in Prolog:

teiler(4, 12).

vorfahr(V, N) :- elternteil(V, N).

vorfahr(V, N) :- elternteil(V, E), vorfahr(E, N).

Einfaches Prolog Programm (Fakten)

Fakten werden durch ein positives Literal (syntaktisch: eine Struktur) ausgedrückt.

```
vater(hans, karin).
```

```
vater(hans, kurt).
```

Hans ist der Vater von Karin. Hans ist der Vater von Kurt.

```
mutter(karla, karin),
```

```
mutter(karla, kurt).
```

Karla ist die Mutter von Karin. Karla ist die Mutter von Kurt.

```
besitzt(hans, buch('Goethe', 'Faust')).
```

Hans besitzt das Buch Goethe, Faust.

Die Bedeutung der Fakten ergibt sich durch ihre Interpretation.

Aus Bequemlichkeit werden Konstanten (Namen) meist klein geschrieben.

Anwendung eines Prolog Programms (Fragen)

Fragen werden durch ein mit ?- eingeleitete Liste von Literalen ausgedrückt. Wenn die Frage keine Variablen enthält lautet die Antwort yes oder no. Bei einer zu bejahenden Frage mit Variablen, wird die Variablenbindung ausgegeben.

```
?- vater(hans, karin) .  
true  
Ist Hans der Vater von Karin? -- Ja
```

```
?- vater(hans, karla) .  
false  
Ist Hans der Vater von Karla? – Nein
```

```
?- vater(V, kurt) .  
V = hans  
Wer ist der Vater von Kurt? – Der Vater heißt Hans.  
(Gibt es ein V, so dass V Vater von Kurt ist? Ja, wenn man V gleich  
Hans setzt.)
```

Prolog ist in der Lage, mehrere Antworten zu geben

Die interaktive Umgebung von SWI-Prolog erwartet auf eine Antwort entweder eine Enter (d.h. das war's) oder ein Semikolon, worauf eine weitere Antwort gesucht wird.

```
?- vater(hans, K) .
```

```
K = karin ;
```

```
K = kurt ;
```

```
No
```

Wie heißt ein Kind von Hans? – Karin ; Kurt

```
?- vater(hans, K), m(K) .
```

```
K = kurt
```

Wie heißt ein männliches Kind von Hans? -- Kurt

Man kann in Prolog mehrere Fragen kombinieren. Prolog gibt nur die Antworten, die komplett alles erfüllen.

Prolog erhält zunächst für `father(hans, K)` die Antwort `karin`, scheitert aber mit `m(karin)` und fragt daraufhin nach einer weiteren Antwort für `father(hans, X)` .

Regeln

Eine Regel (Klausel) besteht aus zwei Teilen, dem Kopf der Regel und durch :- getrennt dem Körper der Regel. Der Kopf der Regel ist genau ein positives Literal, der Körper der Regel enthält eine durch Komma getrennte Liste von negativen Literalen. Die Regel sagt, aus, dass der Kopf der Regel dann erfüllt ist, wenn alle Literale der rechten Seite erfüllt sind (für eine geeignete Belegung der Variablen). Ein Sachverhalt kann durch mehrere Klauseln beschrieben werden. Die einzelnen Klauseln sind durch ‚und‘ verknüpft.

```
grossvater(G, E):- vater(G, V), vater(V, E).
```

```
grossvater(G, E):- vater(G, M), mutter(M, E).
```

G ist der Großvater von E, wenn er entweder der Vater des Vaters V des Enkels E ist oder wenn er der Vater der Mutter M des Enkels E ist.

Wir hätten dies auch anders schreiben können:

```
grossvater(G, E):- vater(G, T), elternteil(T, E).
```

```
elternteil(E, K):- vater(E, K).
```

```
elternteil(E, K):- mutter(E, K).
```

Übung

- 1) Die Datenbasis ist jetzt redundant. Da eine Mutter stets weiblich und ein Vater stets männlich ist. Wir können sie neu schreiben, indem wir Vater und Mutter als Regel definieren und ansonsten einfach geschlechtsneutrale Elternfakten aufschreiben. Wie sieht das Programm aus, wenn wir das konsequent durchführen?
- 1) In Prolog kann man auch rekursive Regeln definieren. Dabei muss man (wie immer) darauf achten, dass neben der Rekursion selbst auch eine geeignete Abbruchbedingung gibt. Drücken Sie mittels Rekursion die Beziehung ‚Vorfahre‘ aus: Eine Person V ist Vorfahre des Nachkommens N, wenn sie Elternteil von N ist oder wenn sie Elternteil eines Vorfahren von N ist.

Aussagenlogik und Prolog-Regeln

Atomare Aussagen heißen auch **Literale**. Steht vor ihnen ein Negationszeichen, so heißen sie **negativ** sonst **positiv**. Formeln der Aussagenlogik lassen sich grundsätzlich in **Konjunktiver Normalform** schreiben:

$$(a \vee b \vee \dots \vee \neg e \vee \neg f) \wedge (r \vee s \vee \dots \vee \neg u \vee \neg v)$$

Bei der Normalisierung ist eine der wichtigsten Umformungen :

$$a \wedge b \wedge c \Rightarrow d$$

wird zu $\neg(a \wedge b \wedge c) \vee d$

und dies wird zu $\neg a \vee \neg b \vee \neg c \vee d$

Formeln in KNF lassen sich in **Klauselform** schreiben: Menge von Und-verknüpften Klauseln. Dabei ist eine Klausel eine Menge von Oder-verknüpften Literalen.

$$\{ \{ a, \neg b, \neg c \}, \{ a, b \}, \{ c \} \}$$

Eine Disjunktion (mit höchstens einen positiven, d.h. nicht-negiertem Literal) heißt **Hornklausel**. In Prolog schreibt man sie als:

$$d :- a, b, c.$$

Dabei soll das – an die Negation der Literale der rechten Seite erinnern.

Ein Prolog-Programm ist die Und-Verknüpfung von Hornklauseln.

Grundbegriffe der (Aussagen-) Logik

Eine logische Formel kann *rein formal* verstanden werden, oder durch eine *Interpretation*.

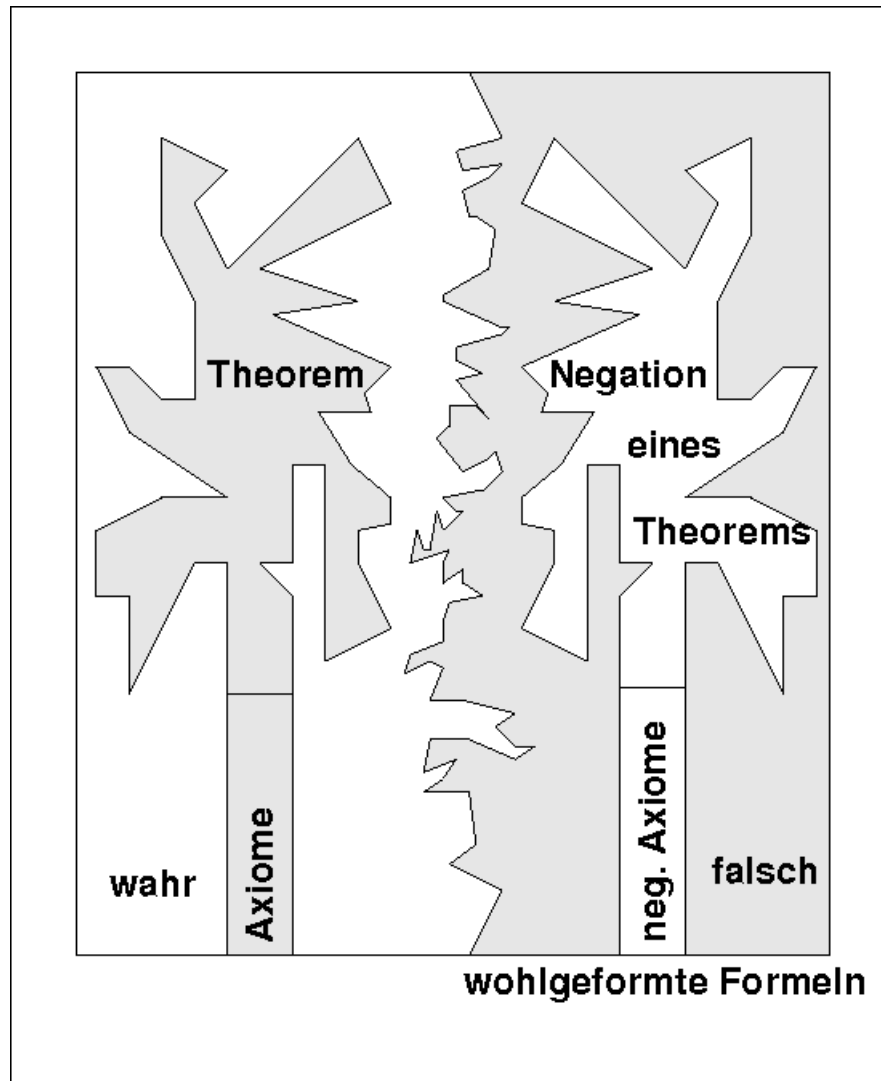
Eine gängige Interpretation ist die Festlegung der Bedeutung von Verknüpfungen und die Belegung von Aussagen mit Wahrheitswerten.

Es gibt Formeln, die bei jeder Belegung wahr sind (**Tautologien**), die bei jeder Belegung falsch sind (**Widerspruch**), die bei bestimmten Belegungen wahr sind (**erfüllbar**) oder die bei bestimmten Belegungen falsch sind (**falsifizierbar**).

Die Untersuchungen auf Tautologien etc. durch **Wahrheitstabeln** erfordert exponentiellen Aufwand (ein Term mehr verdoppelt den Aufwand).

Alternativ kann man Umformungsregeln (**Deduktionsregeln**) festlegen, die es erlauben, aus grundlegenden Formeln (**Axiome**) alle wahren Aussagen herzuleiten. Ein solches formales System bezeichnet man als **Kalkül**.

Theoreme sind ableitbare Aussagen



Strings

Beispiel für die logische Interpretation.

Wir haben folgendes Prolog-Programm:

```
die_sonne_scheint.  
das_wetter_ist_gut:- die_sonne_scheint.
```

In Normalform lautet dies

```
die_sonne_scheint ^ (¬die_sonne_scheint ∨ das_wetter_ist_gut)
```

Aus logischen Aussagen, können wir andere Aussagen herleiten – aber welche?

Es gibt zwei Vorgehensweisen:

1. Ableitung von Folgerungen aus dem Programm (*forward reasoning*)
2. Überprüfen ob sich die Frage aus dem Programm herleiten lässt (*backward reasoning*)

Mit backward reasoning ist es einfacher, eine effiziente Zielorientierung der Herleitung zu erreichen.

In Logiksystemen (nicht in Prolog) wird auch forward reasoning angewendet.

Vorgehensweise von Prolog (Widerspruchsbeweis)

Ein Prolog Programm mag falsch sein, es kann aber solange keinen logischen Widerspruch enthalten, als es nur Fakten und Regeln enthält (und keine negativen Klauseln).

Zwecks Beantwortung einer Frage, wird die Negation der in der Frage enthaltenen Behauptung dem Programm hinzugefügt.

Wenn man zeigen kann, dass das erweiterte Programm widersprüchlich ist, muss das an der negierten Behauptung liegen. Folglich ist die Behauptung erfüllbar.

Diese Vorgehensweise heißt **Widerspruchsbeweis**.

In Prolog lautet die Frage `?-das_wetter_ist_gut`. Das bedeutet, dass wir das „Wetterprogramm“ um `¬das_wetter_ist_gut` erweitern müssen:

```
die_sonne_scheint ^ (¬die_sonne_scheint v das_wetter_ist_gut) ^  
¬das_wetter_ist_gut
```

Jetzt ist die Formel unerfüllbar, d.h. sie ist in sich widersprüchlich. Aus dem Programm folgt, dass das Wetter gut ist.

Erweiterung zur Prädikatenlogik.

Die Prädikatenlogik unterscheidet sich von der Aussagenlogik dadurch, dass ihre Formeln, Variablen und Funktoren enthalten. Nehmen wir das folgende Programm:

```
mensch(sokrates) .  
sterblich(X) :- mensch(X) .
```

Prolog-Variablen gelten immer als all-quantisiert. Das Programm lautet:

```
mensch(sokrates) ∧ ∀x (¬mensch(x) ∨ sterblich(x))
```

oder

```
mensch(sokrates) ∧ ∀x (mensch(x) ⇒ sterblich(x))
```

Die negierte Frage $\neg\text{sterblich}(y)$ führt nicht automatisch zu einem Widerspruch, sondern nur dann, wenn wir $y = \text{sokrates}$ setzen.

Bei der logischen Interpretation von Prologklauseln ist zu beachten, dass alle vorkommenden Variablen all-quantisiert sind.

Das Gegenstück, die existentiell-quantisierten Variablen gibt es in Prolog nicht, sie werden dort durch Konstanten und Funktionen dargestellt (Skolemisierung).

Der Herleitungsmechanismus von Prolog.

Ein Prolog-Programm besteht aus zwei Teilen:

- Dem eigentlichen Programm, bestehend aus Fakten und Regeln
- Der Anfrage, ausgedrückt durch eine negative Horn-Klausel.

Der Herleitungsmechanismus betrachtet die Anfrage als negierte Aussage. Er versucht zu beweisen, dass diese Aussage im Widerspruch zum Programm steht. Da das Programm aus Fakten und Regeln nicht selbst schon widersprüchlich sein kann, muss aus dem Programm dann das Gegenteil dieser Negation, das heißt die vermutete Aussage folgen.

Programm:

```
es_regnet.
```

Frage: regnet es?

```
?- es_regnet.
```

Beweis:

```
es_regnet  $\wedge$   $\neg$ es_regnet ist ein Widerspruch.
```

Also regnet es.

Motivation für die Prolog-Strategie.

Neben Herleitungsregeln benötigt man eine effiziente Strategie, d.h. Vorgehensweise.

Die Prolog-Strategie ist eine Variante der „set of support strategy“. Bei Prolog heißt dies, dass immer nur von der Anfrage oder von den daraus abgeleiteten Aussagen ausgegangen wird.

Eigenschaften und Ziele:

- Durch die Methode des „backward reasoning“ wird die Arbeit auf die Beantwortung der Anfrage fokussiert.
- Die Herleitungsreihenfolge soll leicht verständlich und damit auch per Programm nutzbar sein.
- Nicht leicht „logisch“ formulierbare Zusammenhänge sollen durch eingebaute Prädikate realisiert werden.

Herleitungsverfahren = negatives Resolutionskalkül

Eine (binäre) Resolution bezeichnet die folgende Schlussregel:

$$\begin{array}{l} \neg a \vee b \vee c \\ a \vee x \vee y \\ \hline b \vee c \vee x \vee y \end{array}$$

Der entscheidende Punkt ist, dass in beiden Klauseln ein dasselbe Literal mit unterschiedlichem Vorzeichen ist (hier: a). Es spielt keine Rolle ob b, c, x und y positiv oder negativ sind.

Die Prolog-Strategie geht von der Anfrage (goal) aus.

Sie versucht zunächst das **linkeste Literal** durch Resolution weg zu bekommen (**goal order**).

Es wird die **erste passende Programmklausel** gesucht, die dieses Literal in positiver Form (als Regelkopf) enthält. (**rule order**).

Das Ergebnis der Resolution = Resolvente fungiert in der Folge als neue Anfrage. In der Praxis bedeutet dies, dass zunächst der Körper der Programmklausel ausgewertet wird.

Wenn durch Resolution ein Widerspruch gefunden ist, dann ist die Anfrage bewiesen.

Wenn keine weitere Resolution möglich ist, wird zu dem letzten Resolutionsschritt zurückgegangen, bei der eine weitere Regel zur Auswahl stand und es wird mit dieser Regel ein Beweis versucht (**backtracking**)

Beispiel einer Herleitung

Programm:

R1: a.

R2: b:- c.

R3: b:- a.

Q: ?- a , b. *Frage*

Beweis:

-a, -b // Q
a // **R1**

-b //
b, -c // **R2** *Auswahlpunkt / choice point*

-c
--- fail - *Backtracking: Rückkehr zum letzten Auswahlpunkt*

-b //
b, -a // **R3**

-a
a // **R1**

[] *Widerspruch*

Bewertung

Der Resolutionskalkül ist grundsätzlich korrekt (leitet nie etwas Falsches ab). Er ist im *allgemeinen* auch vollständig – wenn ein Widerspruch besteht, lässt er sich herleiten.

In der vereinfachten Prolog-Form mit genau festgelegter Vorgehensweise ist er aber nicht vollständig! Dies liegt an der Reihenfolgenfolge.

R1: e.
R2: v :- v, e.
R3: v :- e.

R1: e.
R2: v :- e. *Reihenfolge vertauscht*
R3: v :- v, e.

```
?- v.  
   v, -v, -e //R2  
-----  
-v, -e  
   v, -v, -e // R2  
-----  
-v, -v, -e, -e  
...
```

```
?- v.  
   v, -e //R2  
-----  
-e  
   e // R1  
-----  
[]
```

Programming in Logic

Der Vorteil der Vereinfachung und der Vorgehensweise von Prolog liegt in der **Effizienz** und in dem **nachvollziehbaren Ablauf**.

Ein weiterer Vorteil ist die einfache Interpretation von Hornklauseln als **Regeln und Fakten**. Zusammen mit dem Ablauf kann man diese auch als **Programmstrukturen** auffassen.

Der Nachteil liegt in dem **unvollständigen Kalkül**.

Weiterer Nachteil: Ein Prolog-Programm aus positiven Hornklauseln kann **keine Negation** enthalten!

Lösung (**closed world assumption**): was sich nicht beweisen lässt, ist falsch.

Großer Vorteil: der deterministische Ablauf erlaubt es, auch **außerlogische Prädikate**, z.B. für arithmetische Operationen, zu Verwenden.

Einfache außerlogische Prädikate

```
X is 3 * 4.    % Arithmetik
X > 4          % Vergleich (X muss instanziiert sein)
var(X)         % ist erfüllt wenn X nicht instanziiert ist
halt          % beendet Prolog
trace         % schaltet trace ein
listing       % listet die Datenbasis
assertz      % verändert die Datenbasis
```

Beispiel:

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

Unifikation – Umgang mit Variablen

Resolution erfordert, dass Literale (bis auf Vorzeichen) identisch sind. Dazu muss für Variablen eine Ersetzung (Substitution) vorgenommen werden.

```
R1: mensch(sokrates)
```

```
R2: sterblich(X) :- mensch(X)
```

```
Q: ?- sterblich(sokrates)
```

```
-sterblich(sokrates)           // Q  
  sterblich(X), -mensch (X)    // R2, X = sokrates
```

```
-----
```

```
-mensch(sokrates)  
  mensch(sokrates)           // R1
```

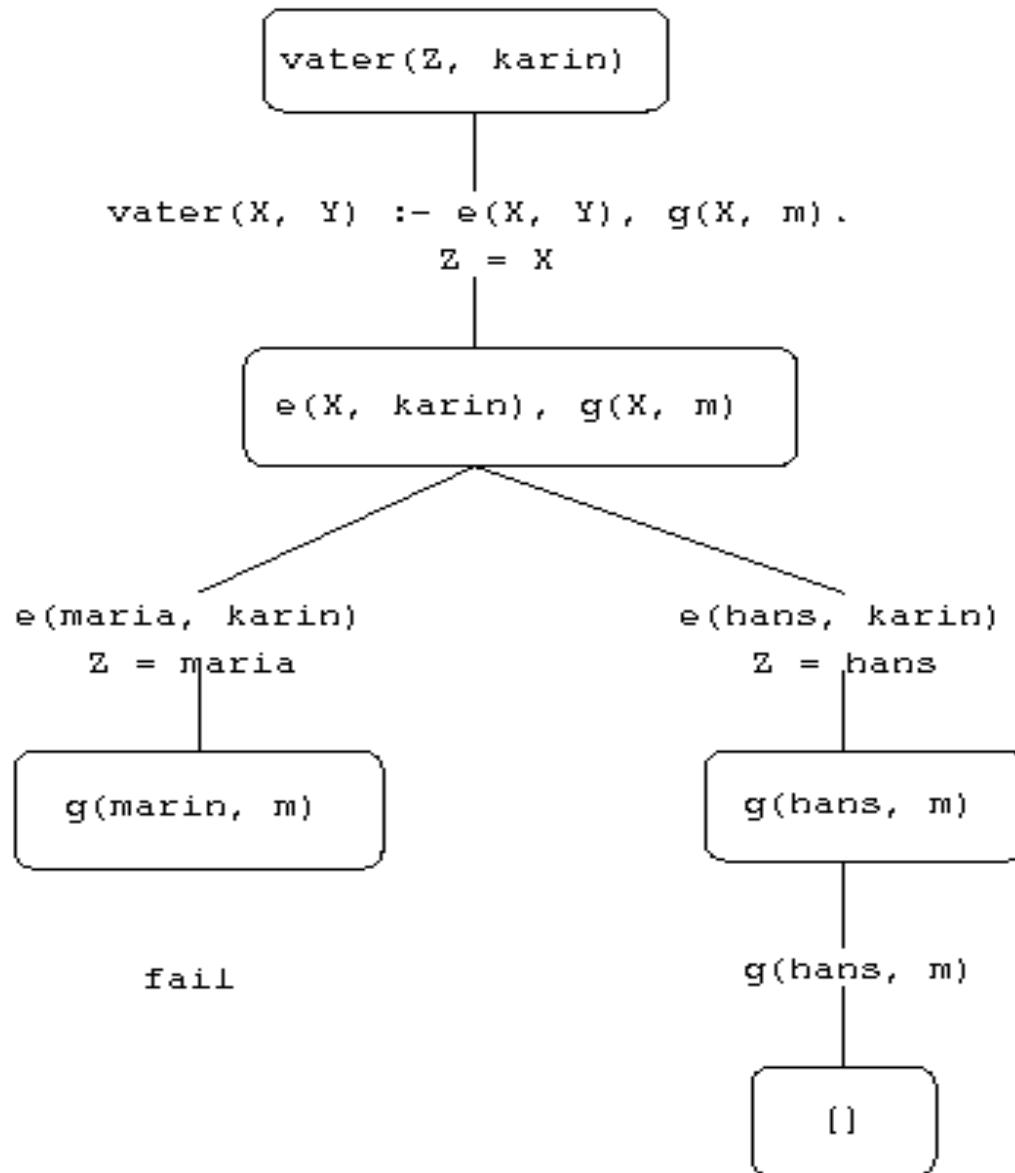
```
-----
```

```
[]
```

Die Unifikation stellt die allgemeinste Substitution dar, die zwei Literale identisch macht. Eine Besonderheit der Unifikation ist, dass damit auch Variablen gleichgesetzt werden können (sharing). Die Unifikation ist keine Zuweisung!

Die Unifikation ist neben dem Backtracking das typische Element eines Prolog-Programms!

Ablauf == Suche im Suchbaum



Cut == Optimierung / prozeduraler Ablauf

Manchmal kann es sinnvoll sein, weitere Suche zu verhindern, weil man weiß, dass keine weitere Lösung gefunden werden kann.

```
% max(X, Y, M) - M ist Maximum von X und Y
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- X < Y.
```

Wenn $X > Y$ ist, braucht man die zweite Klausel nicht mehr zu prüfen. Prolog ermöglicht es, dies mit einem Cut (!) auszudrücken.

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

Eine weitergehende Optimierung, lässt den Vergleich in der zweiten Klausel weg. Dann ist das Programm nicht mehr logisch korrekt!! Es ist jetzt prozedural, d.h. Vom Ablauf abhängig (in Prolog gilt das als schlechter Stil).

```
max(X, Y, X) :- X >= Y, !. % roter Cut
max(_, Y, Y).           % falsch aber im Ablauf kein Problem!
```

Cut - Fail== prozedural definierte Negation

Verneinung (Negation) lässt sich in Prolog nicht logisch beschreiben. Man verwendet entweder implizit die „closed world assumption“ oder benutzt im Lösungsalgorithmus vordefinierte Prädikate. Da der Cut sich die Eigenschaften des Ablaufs zunutze macht, kann man auch damit die Negation „hinbekommen“. Dies wird am Beispiel von Gleichheit (logisch) und Ungleichheit (prozedural) gezeigt

```
% equals(X, Y) - X ist gleich Y (entspricht exakt =)
equals(X, X).
```

```
% notEquals(X, Y) - X ist ungleich Y (entspricht exakt \=)
notEquals(X, X):- !, fail
notEquals(X, Y).
```

Die equals Regel lautet einfach: „zwei Terme die unifizierbar sind, sind gleich“.

Die notEquals Regel ist nur prozedural beschreibbar: „wenn sich herausstellt, dass zwei Terme gleich sind, verhindert man die weitere Suche und meldet, dass man nichts erreicht hat. Andernfalls sagt man, dass die Terme ungleich sind.“

Beide Regeln sind korrektes Prolog. Aber nur equals ist ein Logikprogramm!

Listenoperationen (1)

Konstruktor: `[1, 2, 3]`

Cons-Operation: `[1 | [2, 3]] = [erstes Element | Restliste]`

Allgemeiner: `[1, 2 | Restliste]`

Die Prolog-Regeln ermöglichen diese Operationen in allen Richtungen zu verwenden:

% `anzahl(Liste, N)` - `N` ist die Anzahl der Elemente von `Liste`.

```
anzahl([], 0).
```

```
anzahl([_|Xs], N):-
```

```
    anzahl(Xs, N1),
```

```
    N is N1 + 1.
```

% `ntesElement(N, Liste, E)` - `E` ist das `N`-te Element von `Liste`.

```
ntesElement(1, [X | _], X).
```

```
ntesElement(N, [_ | Xs], Y):-
```

```
    N > 1,
```

```
    N1 is N - 1,
```

```
    n-tesElement(N1, Xs, Y).
```

Es empfiehlt sich, als Kommentar einen „Kopf“ für das Prädikat zu definieren.

Listenoperationen (2)

Regel:

$[] + Bs = Bs$

$[A|As] + Bs = [A|As + Bs]$

% app(As, Bs, ABs) - As + Bs = ABs

app([], Bs, Bs).

app([A|As], Bs, [A|ABs]) :-

app(As, Bs, ABs).

Regel:

$\sim [] = []$

$\sim [A|As] = \sim As + [A]$

% umgekehrt(As, UAs) - Rs = As nur umgekehrt

umgekehrt([], []).

% umgekehrt([X], [X]). % korrekt aber unnötig

umgekehrt([A|As], UAs) :-

umgekehrt(As, Bs),

append(Bs, [A], UAs).

(Prolog-Ablauf von umgekehrt ist $O(n^2)$)

Endrekursionsoptimierung

Was ist mit folgendem Beispiel:

```
% bearbeiteNaechsteAnfrae - ein Server (Beispiel)  
bearbeiteNaechsteAnfrage:-  
    read (Anfrage) ,  
    bearbeite (Anfrage) ,  
    bearbeiteNaechsteAnfrage.
```

Dieses Prädikat ist rekursiv.

Es wäre aber schlecht einen (normalen) Stack aufzubauen (warum?).

Man braucht aber auch keinen Stack!

Viele Compiler sind in der Lage, Endrekursion in Iteration zu übersetzen
(gcc, swipl, scalac, ...)

Vorgehen: Wenn der Algorithmus ein Ergebnis liefern soll, muss dieses auf dem
Hinweg aufgebaut werden. Dazu benötigt man eine Variable (Akkumulator).
Diese muss zuvor initialisiert werden.

Umwandlung Rekursion → Endrekursion

Was ist mit folgendem Beispiel:

```
% anzahl(Liste, N)
anzahl(Liste, N):- anzahl(Liste, 0, N) % Initialisierung

% anzahl(Liste, BisJetzt, N) % endrekursiv
anzahl([], N, N). % bei Abbruch steht Ergebnis fest
anzahl([_|Xs], N0, N):-
    N1 is N0 + 1, % Berechnung auf dem Hinweg
    anzahl(Xs, N1, N).
```

Umkehren der Reihenfolge wird besser ($O(n)$):

```
umgekehrt(Liste, UmgekehrteListe):-
    umgekehrt(Liste, [], UmgekehrteListe).

umgekehrt([], Umgekehrt, Umgekehrt).
umgekehrt([X|Xs], BisJetzt, Umgekehrt):-
    umgekehrt(Xs, [X|BisJetzt], Umgekehrt).
```

Endrekursion ist eine Optimierung (denkt an den Ablauf).

Symbolverarbeitung

Der bequeme Umgang mit Datenstrukturen (+ Möglichkeit eigene Operatorsyntax zu definieren) erleichtert die Anwendung bei Problemen, die durch symbolische Formeln beschreiben werden.

```
:- op(10,yfx,^). % definiert Syntax
:- op( 9, fx,~).
% diff(Formel, Variable, Ableitung)
% Formel wird formal nach Variable abgeleitet.
diff(X, X, 1):- !.
diff(C, X, 0):- atomic(C).
diff(~U, X, ~A):- diff(U, X, A).
diff(U+V, X, A+B):- diff(U.X,A), diff(V,X,B).
...
```

Metaprädikate und Prädikate höherer Ordnung

Metaprädikate erlauben es, per Programm Programmausdrücke zu analysieren. Man kann damit eigene Interpreter schreiben. Die Technik erinnert an „Reflection“ in Java.

Prädikate höhere Ordnung haben Prädikate als Argumente – dies ist nicht mehr die „reine Logik“. Prädikate höhere Ordnung ermöglichen es aber viele Operationen facher auszudrücken. Sie sind vergleichbar mit den Funktionen höherer Ordnung, die in der funktionalen Programmierung eine herausragende Rolle spielen.

```
Term =.. Liste % Liste enthält den Functor gefolgt von den
           % Argumenten des Terms
2 + 3 =.. [+ , 2 , 3]
```

```
atomic(X) % in X ist eine Zahl oder ein Atom
var(X)    % X ist eine (noch) ungebundene Variable
```

```
setof(X, queens(8, X), L)
      % L enthält alle Lösungen des 8-Damen-Problems.
```

Lösungssuche

- Tiefensuche als einfacher Algorithmus
- Breitensuche
- Problemlösen

Tiefensuche

Als Datenbasis haben wir eine Reihe von Fakten oder Regeln, die Kanten in einem (gerichteten) Graphen darstellen. Wir können uns im Beispiel das Prädikat v als direkte Flugverbindung zwischen 2 Flughäfen vorstellen:

```
v(a, b). v(a, c). v(b, d). v(d, e). v(b, e). usw
```

Der Suchalgorithmus soll feststellen, ob es einen Weg von einem Start zu einem Zielort gibt.

```
% es_gibt_Weg(Ort, Ziel)
es_gibt_Weg(Ziel, Ziel).
es_gibt_Weg(X, Ziel):-
    v(X, Y),
    es_gibt_Weg(Y, Ziel).
```

Die zu beantwortende Frage lautet: `?- es_gibt_Weg(a, e).`

Dies ist der Kern des Algorithmus, der aber noch ein paar Verbesserungen braucht.

Ungerichteter Graph 1

Momentan besteht die Beschreibung des Graphen in einer Reihe von Fakten. Am einfachsten kann man diese Beschreibung so erweitern, dass man zu jeder Verbindung $v(x,y)$ auch $v(y,x)$ hinzu nimmt.

Da das etwas mühsam ist, kann man aber statt dessen auch auf die Idee kommen, einfach die „Symmetrie-Regel“ $v(X, Y) :- v(Y, X)$ aufzunehmen.

Dies geht nicht, obwohl es logisch korrekt ist! – Wir haben es wieder mit der Unvollständigkeit von Prolog zu tun.

Der Ausweg:

```
%vs(A, B) -- symmetrische Verbundsbeziehung
```

```
vs(A, B) :- v(A, B).
```

```
vs(A, B) :- v(B, A).
```

```
% es_gibt_Weg(Ort, Ziel)
```

```
es_gibt_Weg(Ziel, Ziel).
```

```
es_gibt_Weg(X, Ziel) :-
```

```
    vs(X, Y),
```

```
    es_gibt_Weg(Y, Ziel).
```

Ungerichteter Graph 2

Auch das funktioniert nicht! Schließlich kann man sich in einen ungerichteten Graphen immer im Kreis bewegen (in gerichteten Graphen mit Kreisen natürlich auch). Kreiswege können unendlich lang werden; wir haben wieder das Problem der Unvollständigkeit.

Auswege:

- Markieren der besuchten Knoten (scheidet in dem Beispiel aus)
- Merken der Knoten in einer Liste.

```
% es_gibt_Weg(Ort, Ziel)
es_gibt_Weg(Start, Ziel):-
    es_gibt_Weg(Start, Ziel, [Start]).

% es_gibt_Weg(Ort, Ziel, Besucht)
es_gibt_Weg(Ziel, Ziel, _).
es_gibt_Weg(X, Ziel, Besucht):-
    vs(X, Y),
    % nicht member(Y, Besucht)
    es_gibt_Weg(Y, Ziel, [Y | Besucht]).
```

Problem Verneinung

Prolog kann Verneinung nicht über eine Hornklausel ausdrücken (wenn wir das negative Literal *member* verneinen, haben wir zwei positive Literale)!

Es gibt allerdings zwei Möglichkeiten:

- Eingebautes Prädikat zur Verneinung `\+`
- Beeinflussung der Suche mittels `!`, `fail`.

Verneinung bedeutet hier: das Ziel lässt sich nicht beweisen.

```
% es_gibt_Weg(Ort, Ziel)
es_gibt_Weg(Start, Ziel):-
    es_gibt_Weg(Start, Ziel, [Start]).

% es_gibt_Weg(Ort, Ziel, Besucht)
es_gibt_Weg(Ziel, Ziel, _).
es_gibt_Weg(X, Ziel, Besucht):-
    vs(X, Y),
    \+ member(Y, Besucht),
    es_gibt_Weg(Y, Ziel, [Y | Besucht]).
```

Rückgabe des durchlaufenen Weges

Nachdem der Algorithmus funktioniert, besteht die Lösung in einer einfachen Erweiterung. Es ist nur nötig auf die richtige Reihenfolge zu achten.

```
% es_gibt_Weg(Ort, Ziel, Weg)
es_gibt_Weg(Start, Ziel, Weg):-
    es_gibt_Weg(Start, Ziel, [Start], Weg).

% es_gibt_Weg(Ort, Ziel, Besucht)
es_gibt_Weg(Ziel, Ziel, _, [Ziel]).
es_gibt_Weg(X, Ziel, Besucht, [Y | Weg]):-
    vs(X, Y),
    \+ member(Y, Besucht),
    es_gibt_Weg(Y, Ziel, [Y | Besucht], Weg).
```

Das Beispiel macht deutlich, dass sich die Tiefensuche in Prolog ziemlich einfach realisieren lässt.

Als nächstes könnten wir den Graphen mit Gewichten versehen und nach kürzesten Wegen suchen. Dazu können wir entweder den Dijkstra-Algorithmus (Variante der Breitensuche) oder aber die Tiefensuche für alle möglichen (endlichen) Wege verwenden. Für Letzteres benötigen wir aber weitere Hilfsprädikate.

`es_gibt_Weg(b, X, [b], Weg)`

`Z ← b`
`X ← b`
`Weg ← [b]`

`K ← b`
`Z ← X`
`B ← [b]`
`Weg ← [b|W]`

`es_gibt_Weg(Z, Z, _, [Z])`

`es_gibt_Weg(K, Z, B, [K|W]) :-`
`vs(K, N),`
`\+ member(N, B),`
`es_gibt_Weg(N, Z, [N|B], W).`



Prädikate für Lösungsmengen

```
bagof(Template, Ziel, Menge)  
setof(Template, Ziel, Menge)
```

Ziel ist die zu untersuchende Zielfrage. *Template* ist eine Variable oder eine Struktur von Variablen. *Menge* enthält die Menge aller Templates mit deren Variablen sich das Ziel beweisen lässt.

bagof kann dieselbe Lösung mehrfach enthalten (wenn sie auf verschiedenem Wege gefunden wurde). Bei *setof* kommt jede Lösung nur einmal vor. Wenn keine Lösung gefunden wird, scheitern beide Prädikate.

```
?- setof(W, es_gibt_Weg(a, e, W), Wege).
```

Gibt die Liste *Wege* aller gefundenen Wege zurück.

```
?- setof((W, Ziel), es_gibt_Weg(a, Ziel, W), Wege).
```

Gibt alle Wege zu allen möglichen Zielen ab *a* zurück.

Breitensuche 1

Bei der Breitensuche werden mehrere mögliche Wege gleichzeitig gesucht. In dieser ersten Lösung wird daher der Parameter des Ausgangsortes durch eine Liste von Orten ersetzt (später, wenn man die Wege kennen will, braucht man eine Liste von Wegen).

Auf Kreise wird zunächst nicht getestet. Dies führt (nur) bei vergeblicher Suche zu unendlichem Ablauf!

```
% es_gibt_Weg_BS(Ort, Ziel)  
es_gibt_Weg_BS(Start, Ziel):-  
    es_gibt_Weg([Start], Ziel).
```

```
% es_gibt_Weg(Ort, Ziel)  
es_gibt_Weg([Ziel | _], Ziel).  
es_gibt_Weg([X | Xs], Ziel):-  
    nachbarn(X, Ns),  
    append(Xs, Ns, Ys),  
    es_gibt_Weg(Ys, Ziel).  
es_gibt_Weg([_|Xs], Ziel):-  
    es_gibt_Weg(Xs, Ziel).
```

```
% nachbarn(Knoten, Nachbarknoten)  
nachbarn(X, Ns):-  
    setof(N, vs(X, N), Ns).
```

Breitensuche 2

Um den Weg zu bekommen, nehmen wir die Liste der Knoten auf. Anstelle einer Variablen/Konstanten Knoten, verwenden wir [Knoten, VorgaengerKnoten, ...], d.h. die Liste der Knoten in einer Reihenfolge die umgekehrt zum gesuchten Weg ist.

```
% es_gibt_Weg(Start, Ziel, Weg)
es_gibt_Weg(Start, Ziel, Weg):-
    es_gibt_Weg_BS([[Start]], Ziel, Weg).

% es_gibt_Weg_BS(Pfade, Ziel, Weg)
es_gibt_Weg_BS([[Ziel|Ks]|_], Ziel, Weg):-
    reverse([Ziel|Ks], Weg). % reverse ist vordefiniert
es_gibt_Weg_BS([P|Ps], Ziel, Weg):-
    nachbarn(P, Ns),
    append(Ps, Ns, Ys),
    es_gibt_Weg_BS(Ys, Ziel, Weg).
es_gibt_Weg_BS([_|Ps], Ziel, Weg):-
    es_gibt_Weg_BS(Ps, Ziel, Weg).

% nachbarn(Knoten, Nachbarknoten)
nachbarn([[X|Xs], Ns):-
    setof([N, X | Xs], vs(X, N), Ns).
```

Breitensuche 3

Jetzt soll die Suche noch so erweitert werden, dass Kreise vermieden werden. Dazu muss nur die Suche nach Nachbarknoten so verändert werden, dass kein Knoten ausgewählt wird, der längs dem bisher betretenen Pfad liegt.

```
% nachbarn(Knoten, Nachbarknoten)
nachbarn([[X|Xs], Ns):-
    setof([N, X | Xs], neuer_Nachbar(X, N, [X|Xs]), Ns).

% neuer_Nachbar(Knoten, Nachbar, Bekannte)
%   Nachbar ist ein Nachbar von Knoten aber noch nicht in der Menge der
%   bekannten Knoten
neuer_Nachbar(Knoten, Nachbar, Bekannte):-
    vs(Knoten, Nachbar),
    \+ member(Nachbar, Bekannte).
```

Anmerkung: Es ist in Prolog auch möglich, direkt in *setof* eine Konjunktion von Zielen anzugeben. Hier geht es aber mehr um die Einfachheit und Lesbarkeit.

Problemlösen.

Viele Aufgaben lassen sich durch Varianten von Tiefensuche oder Breitensuche lösen. Eine besondere Variante ist die kombinatorische Suche. In der einfachsten Form verläuft diese Variante nach der folgenden Form (generate and test):

```
problem_loesung(Loesungs_Parameter) :-  
    generiere(Loesungs_Parameter) ,  
    teste(Loesungs_Parameter) .
```

Der Test erzwingt solange ein Backtracking, bis eine akzeptable Lösung erzeugt wurde.

Hilfsprädikate zum Lösungsgenerieren.

Häufig besteht das Generieren einer Lösung darin, dass man aus einer vorhandenen Menge von Elementen die richtigen (in geeigneter Reihenfolge ausgewählt). Am verbreitetsten sind:

```
% member(X, Xs)
%   X ist Element der Liste der Xs
% select(X, Xs, Rs)
%   X ist Element der Xs und Rs ist die Liste ohne X.
```

Eine mögliche Anwendung ist die Definition eines Prädikats, das Permutation, d.h. beliebige Vertauschungen der Reihenfolge von Elemente beschreibt:

```
% permutation(Liste, Permutierte_Liste)
%   Permutierte_Liste ist eine Permutation von Liste.
```

```
permutation([], []).
permutation(Xs, [X | Zs]):-
    select(X, Xs, Ys),
    permutation(Ys, Zs).
```

Naives generate & test

Die einfachste Anwendung des generate & test – Prinzips kann man an einem Algorithmus zum Sortieren einer Liste von Zahlen verdeutlichen:

```
% sortiert(Xs, Xs_sortiert)
sortiert(Xs, Xs_sortiert):-
    permutation(Xs, Xs_sortiert),
    aufsteigend_geordnet(Xs_sortiert).

% aufsteigend_geordnet(Xs)
aufsteigend_geordnet([]).
aufsteigend_geordnet([X]).
aufsteigend_geordnet(X, Y | Xs):-
    X =< Y,
    aufsteigend_geordnet([Y | Xs]).
```

Laufzeitkomplexität: $O(e^n)$!!

Verbessertes generate & test

Die Verbesserung von generate & test besteht darin, möglich nur aussichtsreiche Kandidaten zu generieren und dann zu untersuchen. Das Ausmaß der Verbesserung hängt vom Problem ab. Beim Sortieren kann man extrem viel erreichen; bei anderen Problemen wird man aber trotz Verbesserung bei exponentieller Komplexität bleiben.

Die beiden folgenden Folien geben zwei Varianten des n -Damen-Problems wieder, einmal in ganz naiver Form und einmal in der verbesserten Form (aber immer noch in $O(e^n)$).

Damenproblem: Platziere auf einem quadratischen Schachbrett mit $n \times n$ Feldern n Damen so, dass keine die andere schlagen kann. Damen können andere Damen schlagen, wenn diese in der gleichen horizontalen Reihe, vertikalen Spalte oder in diagonaler Richtung stehen.

Darstellung: die n -Damen werden durch eine Liste von n -Zahlen dargestellt. Die jeweilige Zahl gibt eine Zeilennummer wieder und die Position der Zahl in der Liste die Spaltennummer. Die Zahlen werden von 1 bis N gezählt.

Hilfsprädikate für das Damenproblem

```
% safe(Qs) die Plazierung der Damen in Qs ist sicher.
safe([]).
safe([Q|Qs]):- safe(Qs), \+ attack(Q, Qs).

% attack(Q, Qs) die Dame Q kann durch eine Dame aus Qs geschlagen
werden.
attack(Q, Qs):- attack(Q, 1, Qs).

attack(X, N, [Y|_]):- X is Y + N .
attack(X, N, [Y|_]):- X is Y - N .
attack(X, N, [_|Ys]):- N1 is N + 1, attack(X, N1, Ys).

% permutation(Xs, Ys) wie gehabt

% range(A, B, Zs)
% Zs ist die Liste der ganzen Zahlen von A bis B.
range(N, N, [N]).
range(M, N, [M|Ns]):- M < N,
    M1 is M + 1, range(M1, N, Ns).
```

Naive Lösung

```
% queens(N, Queens)
% Queens ist eine Plazierung, die das N-Damen-Problem löst. Es ist
% dargestellt als gesuchte Permutation der Zahlen von 1 bis N.
```

```
queens(N, Qs):-
    numlist(1, N, Ns),
    permutation(Ns, Qs),
    safe(Qs).
```

Bessere Lösung (klassisches Backtracking)

```
% queens(N, Queens)
% Queens ist eine Plazierung, die das N-Damen-Problem löst. Es ist
% dargestellt als gesuchte Permutation der Zahlen von 1 bis N.

queens(N, Qs):-
    numlist(1, N, Ns),
    queens(Ns, [], Qs).

queens(UnplacedQs, SafeQs, Qs):-
    select(Q, UnplacedQs, UnplacedQs1),
    \+ attack(Q, SafeQs),
    queens(UnplacedQs1, [Q|SafeQs], Qs).
queens([], Qs, Qs).
```

Die Verbesserung besteht darin, dass der Lösungsvektor nach und nach aufgebaut wird und in jedem Schritt sofort geprüft wird, ob die Auswahl zu einer Lösung führen kann. Während das „dumme“ Verfahren zunächst n Zahlen bestimmt und erst am Ende prüft.