Programmierung in Prolog

Zur „Übersetzung“ bekannter Programmstrukturen benötigen wir:

• Elementare Werte	Zahlen, Atome
• Datenstrukturen:	Strukturen und Listen
• Variablen:	einmalig festzulegende Variable
• Ausdrücke	Strukturen
• Fallunterscheidung:	Pattern-Matching
• „Wiederholung“:	Rekursion und Backtracking
• Elementare Anweisungen:	Aussagen, eingebaute Prädikate
• Prozeduren:	Prädikate (Fakten und Regeln)
• Ein- / Ausgabe usw:	eingebaute Prädikate

Fallunterscheidung erfolgt durch mehrere Regeln, die zu unterschiedlichen Aufrufen passen und durch weitere Bedingungen definiert sind.

Beispiel: Auswertung von Formeln:

```
% eval(Formel, Ergebnis)
eval(X + Y, Z):-
  eval(X, X1), eval(Y, Y1), Z is X1 + Y1.
eval(X * Y, Z):-
  eval(X, X1), eval(Y, Y1), Z is X1 * Y1.
eval(X, X):- numeric(X).
```

Beispiel: Berechnung der Fibonacci-Funktion:

```
% fib(N, Fib_N)
fib(0, 0).
fib(1, 1).
fib(N, F):- N > 0,
  N1 is N – 1, fib(N1, F1),
  N2 is N – 2, fib(N2, F2),
  F is F1 + F2.
```

Algebraische Datentypen beschreiben die Struktur der Daten

(es gibt Prolog-Varianten mit Typdeklaration: Visual-Prolog, Mercury)

Baum = +(Baum, Baum) | *(Baum. Baum) | Zahl

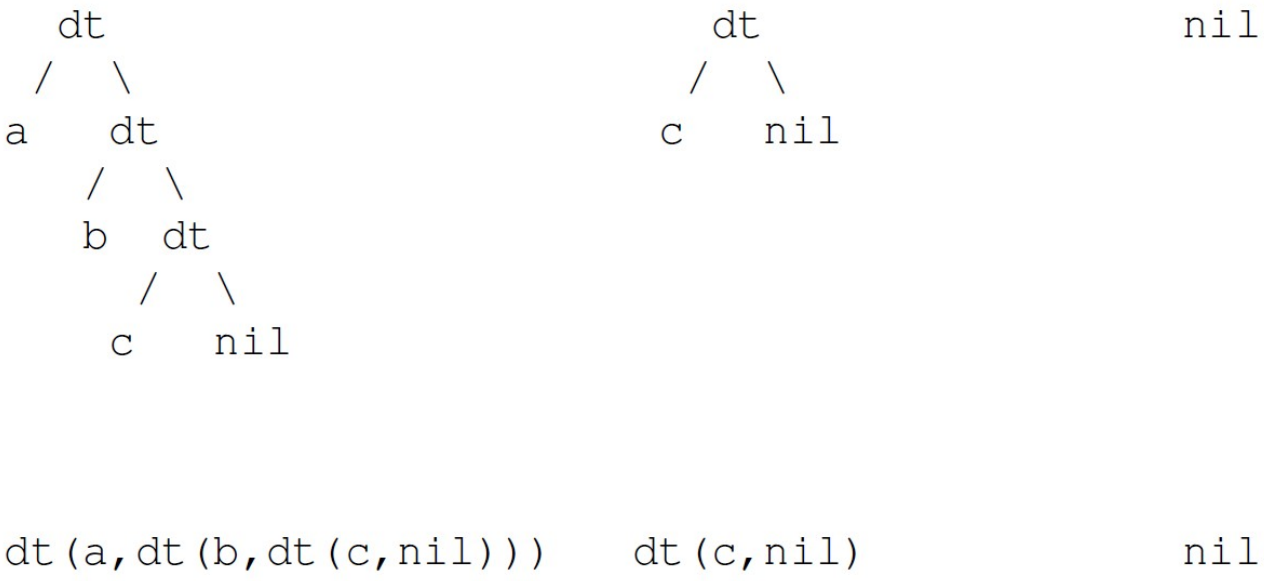
Oder

Baum = Baum + Baum | Baum * Baum | Zahl

In Prolog kann man Typen durch Typprädikate verdeutlichen – und überprüfen.

```
% baum(Baum)
%  Baum ist ein korrekter Baum
baum(Links + Rechts):-
    baum(Links),
    baum(Rechts).
baum(Links * Rechts):-
    baum(Links),
    baum(Rechts).
baum(Zahl):-
    number(Zahl).
```

Example of lists




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

Listenoperationen (1) (unveränderliche Datenstruktur)

Algebraische Definition: **`Liste = [] | .(Element, Liste)`**

Prolog definiert eine gut lesbare Syntax:

Leere Liste: `[]`
Konstruktor: `[1, 2, 3] == .(1, .(2, .(3, [])))`
Cons-Operation: `[A | B] == .(A, B)`
`[1 | [ 2, 3]] = [`*`erstes Element | Restliste`*`]`
Allgemein: `[1, 2 |` *`Restliste`*`]`

`% isList(L) –` *L ist eine Liste (in Prolog eingebaut:* `is_list`*)*

```
isList([]).
isList([_|Xs]):- isList(Xs).
```

Diese Operationen können bei Konstruktion und bei Pattern-Matching angewendet werden.

`% summe(Liste, S) – S ist die Summe der Listenelemente`

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

?- A = [1,2], B = [7 | A], summe(B, C).
A = [1,2], B = [7, 1, 2], C = 10
```

Listenoperationen (2)

Regel:

```
[] ::: Bs = Bs        (::: = append-Operator von Scala)
[ A | As ] ::: Bs = [ A | As ::: Bs ]
```

```
% append(As, Bs, ABs) - As ::: Bs = ABs
append([], Bs, Bs).
append([A|As], Bs, [A|ABs]):- append(As, Bs, ABs).
```

Regel:

```
reverse [] = []
reverse [ A | As ] = reverse As ::: [ A ]
```

```
% reverse(As, UAs) - UAs = As nur umgekehrte Reihenfolge
reverse([], []).
reverse([A|As], UAs):-
    reverse(As, Bs),
    append(Bs, [A], UAs).
```

(Ablauf von append ist O(n), Ablauf von *reverse* ist $O(n^2)$)

Endrekursionsoptimiertung

Wie vermeidet man die Nachteile der Rekursion (Speicherverbrauch)?

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

Dieses Prädikat ist rekursiv.
Es wäre aber schlecht, einen (normalen) Stack aufzubauen.
Man braucht aber auch keinen Stack, da es keinen notwendigen Rückweg gibt!!!

Der Prolog-Compiler übersetzt Endrekursion in Iteration.
(das gilt auch für gcc -O2, für clang und für scalac usw. für viele andere Compiler – außer Java)

Umwandlung von Normalrekursion in Endrekursion:
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

```
% summe(Liste, S)
summe(Liste, S):- summe(Liste, 0, N)  % Initialisierung

% summe(Liste, BisJetzt, S)           % endrekursiv
summe([], S, S).                      % bei Abbruch steht Ergebnis fest
summe([X|Xs], S0, S):-
    S1 is S0 + X,                     % Berechnung auf dem Hinweg
    summe(Xs, S1, S).
```

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

```
reverse(List, Reversed):-
    reverse(List, [], Reversed).

reverse([], Reversed, Reversed).
reverse([X|Xs], SoFar, Reversed):-
    reverse(Xs, [X|Sofar], Reversed).
```

Endrekursion ist eine Optimierung (denkt an den Ablauf).

reverse ist jetzt O(n), da append O(n) durch cons O(1) ersetzt wurde.
(cons = `[ | ]`*)*

Symbolverarbeitung *

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

Prädikate höherer Ordnung

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 vergeichbar mit den Funktionen höherer Ordnung, die in der funktionalen Programmierung eine herausragende Rolle spielen.

```
call(F, Arg1, Arg2) % Aufruf eines unbekannten Prädikats

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

Abgrenzung: Metaprädikate: Metaprädikate erfragen die Eigenschaften von Objekten (wie Reflection in Java):

```
var(X).
integer(X).
```

Prädikate höherer Ordnung ergeben eine höhere Abstraktionsebene

```
% map(Xs, F, Ys)
% F(X,Y) ist eine Funktion, Ys ist die Liste der
% Funktionswerte der Elemente von Xs.
map([], F, []).
map([X|Xs], F, [Y|Ys]):- call(F, X, Y), map(Xs, F, Ys).

quadrat(X, Y):- Y is X * X.
plus(X, Y, Z):- Z is X + Y.

?- map([1,2,3,4], quadrat, Ys).  Ys = [1,4,9,16]
```

... es ist sogar einfach, anonyme Funktionen zu definieren
(Prolog als Interpreter eigener Sprachen)

SWI-Prolog verfügt über einige Prädikat zweiter Ordnung (u.a. map_list).

In der funktionalen Programmierung spielen Funktionen höherer Ordnung eine ganz erhebliche Rolle.

„Intelligente“ Anwendungen: Lösungssuche

- Tiefensuche als einfacher Algorithmus(Backtracking)
- (Breitensuche ist natürlich auch möglich)
- Problemlösen

Spezifikation und Ablauf (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 der Tiefensuche 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).
```

Problem Verneinung

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

Wir müssen die Verneinung prozedural ausdrücken.

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), % not member … prozedurale Verneinung
   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.

Prädikate für Lösungsmengen

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

Ziel ist die zu untersuchende Zielanfrage. *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 gefungen 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.

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 auswählt). 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.
```

Aufgabe: wie lautet select?

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 | Ys]).
```

Laufzeitkomplexität: $O(e^n)$!!

Verbessertes generate & test

Die Verbesserung von generate & test besteht darin, möglich nur ausichtsreiche 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: Plaziere auf einem quadratischen Schachbrett mit *n* x *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.

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.

Backtracking in Java

Backtracking: rekursives Ausprobieren mit Zurückgehen bei Scheitern == Tiefensuche

```
/**
 * Loest das N-Damen Problem.
 *
 * @param qs Array der Damen-Positionen.
 * @param col Anzahl der bereits gesetzen Spalten.
 * @return true wenn es eine Loesung gibt.
 */
static boolean queens(int[] qs, int col) {
    if (col == qs.length)
        return true;
    // finde die erstbeste passende Zeile (row)
    for (int row = 0; row < qs.length; row++) {
        if (notAttacked(qs, row, col)) {
            qs[col] = row;
            // wenn auch der Rest gesetzt werden kann,
            // sind wir fertig
            if (queens(qs, col+1)) return true;
        }
    }
    return false;
}
```

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, Xs)
%   Xs ist die Liste der ganzen Zahlen von A bis B.
range(A, B, Xs):- bagof(X, between(A, B, X), Xs).
```

„Lehren“ aus der Logikprogrammierung

- Eine **logische Spezifikation** stellt keinen Ablauf dar (ist kein Algorithmus)
- Algorithmen basieren auf logischen Aussagen (Zusicherungen) und der Festlegung des **Lösungswegs** (Ablauf)
- **Fallunterscheidungen** sind durch Bedingungen (**Guard**) geschützt
- **Backtracking** realisiert eine effiziente Tiefensuche.
- Backtracking/Tiefensuche ist **nicht vollständig**
- Rekursion **(Endrekursion)** kann effizient programmiert werden!
- Auf der Basis von **algebraischen Datentypen** lassen sich Bedingungen durch **Pattern-Matching** ausdrücken.

Die Mechanismen von Prolog sind:

- **Unifikation** = Variablenersetzung und Mechanismus des Pattern-Matchings
- **Resolution** = Ersetzung von Ziel durch Regelkörper (Aufruf)
- **Lösungsstrategie** = Tiefensuche (SLD-Resolution, mit Backtracking) basierend auf Zielreihenfolge (links → rechts) und Regelreihenfolge (oben → unten)