Built-In Prädikate

Listen und Listenmanipulation

```
?- [surfers, do, it, standing, up] = [A,B | [C | D]].  %Kopf-Rest-Separator

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

Enthaltensein eines Elementes:	member(Element, Liste)
Bestimmung des letzten Elementes:	last(Element, Liste)
Aneinanderreihen von Listen:	append(L1, L2, L1undL2)
Bestimmung des I-ten Elementes:	nth0(Index,Element,Liste) bzw. nth1/3
Entfernen aller Vorkommen :	delete(Liste1, Element, Liste1oElement)
Spiegelung einer Liste:	reverse(L1, L2)
Sortieren zweier Listen:	merge(L1,L2,L1L2sortiert)

Definition eigener Operationen

op(Praezedenz, Typ, Name)

Praezedenz: zwischen 1 und 1200, wobei Operatoren mit kleinerer Präzedenz Unterstrukturen von Operatoren mit höherer Präzedenz bilden.

Typ: Infixtypen: xfx xfy yfx
Prefixtypen: fx fy
Postfixtypen: xf yf

x besagt, dass alle in diesem Argument vorkommenden Operatoren eine kleiner Präferenz als der Operator f haben müssen.
y besagt, dass alle ... eine gleiche oder kleinere Präferenz als f haben müssen

Bsp.

\+ ist vom Typ yfx, daher kann 3+4 nur als +(+(1,2)3) interpretiert werden

:- ist vom Typ xfx, daher ist a :-b :- c nicht zulässig

not ist vom Typ fy, sonst wäre die Formulierung not(not(p)) nicht zulässig

Programmkontrolle

- Anordnung von Klauseln und Literalen
- Verhindern des Backtracking
- Negation als Scheitern
- Ablaufsteuerung (fail,true,repeat, ...)
- *Debugger (trace, spy, debug,...) -> Ref 3.39*
- *Fehlerbehandlung -> Ref. 3.8*

Anordnung von Klauseln

Bsp.

```
letztes([K|R],E) :- letztes(R,E).
letztes([E],E).

?- letztes([1,2,3],3).
yes
?- letztes(L,3).
#
```

```
letztes([E],E).
letztes([K|R],E) :- letztes(R,E).

?- letztes(L,3).
L = [3]
```

Die Reihenfolge der Klauseln hat Einfluß auf den Erfolg eines Beweises in Prolog! (trotz logischer Äquivalenz)

Tip: Spezialfälle vor allgemeineren Klauseln

Anordnung der Literale

Bsp. Ist eine Zahl größer oder gleich einer anderen?

```
groesser_gleich(X,Z) :- addiere_1(Y,X), groesser_gleich(Y,Z).
groesser_gleich(X,X).
addiere_1(0,1).

?- groesser_gleich(G,0).
G=1
```

aber:

```
groesser_gleich(X,Z) :- groesser_gleich(Y,Z), addiere_1(Y,X).
groesser_gleich(X,X).
addiere_1(0,1).
```

führt bei gleicher Anfrage zu einer Endlosschleife.

Linksrekursive Regeln, vor denen keine Fakten für das gleiche Prädikat stehen, enthalten in Prolog die Gefahr einer Endlosschleife.

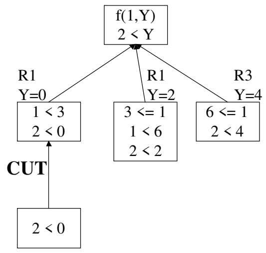
Verhinderung des Backtracking

Bsp.
f(X,0) :- X < 3.
f(X,2) :- 3 =< X, X<6.
f(X,4) :- 6 =< X.

?- f(1,Y), 2 < Y.
no

oder

f(X,0) :- X < 3, !.
f(X,2) :- 3 =< X, X<6, !.
f(X,4) :- 6 =< X, !.



Definition des CUT-Mechanismus

- Steht der CUT am Ende einer Regel und das System erreicht den CUT, dann ist das „Kopfprädikat" erfolgreich und beim Backtracking werden keine Alternativen gesucht.
- Stehen hinter dem CUT noch weitere Ziele, so führt ein Backtracking innerhalb der Regel nur bis zum CUT und das „Kopfprädikat" ist falsch.

Negation as Failure

Bsp. „Maria mag alle Tiere außer Schlangen"
mag(maria,X) :- schlange(X),!,fail
mag(maria,X) :- tier(X).
scheitert immer

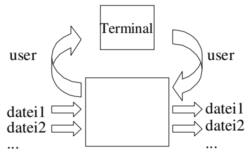
Def. von not(P): Wenn P wahr, dann scheitert not(P), ansonst ist not(P) wahr.
not(P) :- P,!,fail.
not(P) :- true

mag(maria,X) :- tier(x), not schlange(X).

NOT ist mit Vorsicht einzusetzen!!

r(a). q(b). p(X) :- not r(X).
?- q(X), p(X).   X=b
?- p(X), q(X).   no

Ein-/Ausgabe-Formatierung



see(Datei): lenkt den aktuellen Eingabestrom auf Datei um
tell(Datei): dito für Ausgabestrom
seen bzw. told: schließen die aktuellen Ein-/Ausgabedatei
get(Zc), get0(Zc), put(Zc): zeichenweise Verarbeitung
read(Term), write(Term): Verarbeitung von Termen

Bsp. zur Ein-/Ausgabe

Bsp. Mensch-Maschine-Interaktion
action :- write('Naechste Zahl bitte: '), read(X), bearbeite(X).

Bsp. Anzeigen von Listen
schreibe_liste([]).
schreibe_liste([X|L]) :- write(X), nl, schreibe_liste(L).

Bsp. Ausgabe in eine Datei
schreib_term(Datei) :- repeat, nl, read(X),term_add(Datei,X).
term_add(Datei) :- X == end_of_file, close(Datei).
term_add(Datei,X) :- tell(Datei), writeq(X), write('.'),nl,tell(user),fail.

Termmanipulation

Typprüfung:

var(X) / nonvar(X)	X ist z.Z. eine freie / gebundene Variable
atom(X)	X ist ein Atom
integer(X)	X ist eine ganze Zahl
atomic(X)	X ist eine ganze Zahl oder ein Atom

Bsp. Wertzuweisung
..., Z is X + Y, ...
..., integer(X), integer(Y), Z is X + Y, ... % „Bewachung" des Prädikats Z is X*Y

Bsp. Zähle das Auftreten eines Atoms in einer Liste
zaehle(_,[],0).
zaehle(A, [A | L], N) :- !, zaehle(A, L, N1), N is N1 + 1.
zaehle(A, [_ | L], N) :- zaehle(A,L,N).

Termzerlegung/-konstruktion

Struktur =.. [Funktor | Argumentliste]	die Liste enthält den Hauptfunktor der Struktur und ihrer Argumente
funktor (Struktur, Funktor, Stelligkeit)	Funktor und Stelligkeit bzgl. Struktur
arg(N, Struktur, Argument)	N-te Argument von Struktur
name(Atom, Codeliste)	Codeliste der ASCII-Zeichen von Atom

Bsp. geometrische Figuren vergrößern

```
quadrat(Seite)
dreieck(Seite1, Seite2, Seite3)
kreis(Radius)
...
vergroessere(Fig, F, Fig1) :-
  Fig =.. [Typ | P],
  multipliziere_liste(P, F, P1),
  Fig1 =.. [Typ | P1].
```

```
multipliziere_liste([], _, []).
multipliziere_liste([X|L], F, [X1|L1]) :-
  X1 is F*X,
  multipliziere_liste(L,F,L1).
```


Manipulation der Datensammlung

asserta(Klausel)	füge Klausel am Anfang ein
assertz(Klausel)	füge Klausel am Ende ein
retract(Klausel)	entferne Klausel, die mit Klausel unifizierbar ist
clause(hd, body)	sucht nach Klauseln mit der Struktur hd als Kopf und body als Rumpf

Bsp. Wetter

```
schoen :- sonnenschein, not regen.
abscheulich :- regen, nebel.
regen.
nebel.
warm.
```

```
?- retract(nebel).
yes
?- abscheulich.
no
?- asserta((schoen :- warm, regen.)).
yes
?- schoen.
yes
```


Sammeln von Daten

bagof(X, P, L)	L ist eine Liste aller X, die P erfüllen
setof(X, P, L)	L ist eine sortierte Liste aller X, die P erfüllen (ohne Duplikate)
findall(X, P, L)	wie bagof, jedoch ungeachtet der Variablenbindung in P

Bsp.

```
klasse(a, vok).
klasse(b, kon).
klasse(c, kon).
klasse(d, kon).
klasse(e, vok).
klasse(f, kon).
```

```
?- bagof( Bs, klasse(Bs, kon), Bs).
Klasse = vok
Bs = [a,e];

Klasse = kon
Bs = [b, c, d, f]

?- setof( Kl/Bs, klasse(Bs, Kl), Liste).
Liste = [kon/b, kon/c, kon/d, kon/f, vok/a, vok/e]

?- findall( Bs, klasse(Bs, Kl), Bs).
Bs = [a, b, c, d, e, f]
```

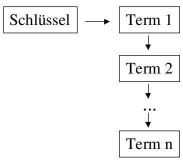
Implementierung von findall

```
findall( X, Ziel, Xliste) :-
  call(Ziel),                % finde eine Loesung
  assertz( stack(x) ),       % fuege sie in die Datenbasis ein
  fail                       % loese Backtracking aus => weitere Loesungen
  ; assertz( stack(ende) ),  % markiere das Ende der Loesungen
  sammle( Xliste ).          % sammle die Loesungen

sammle(L) :-
  retract( stack(X) ),!,         % entferne naechste Loesung
  (X == ende,!,L=[];             % Ende der Loesungen
   L = [X|Rest], sammle(Rest)    % ansonsten: sammle Rest
  ).
```


Exkurs: Interne Datensammlung

(Ablage von Zwischenergebnissen)



recorda(+key, +term, -db_ref_nr)	reiht term als erstes Element der unter key abgespeicherten Terme ein
recorded(+key, -value, -db_ref_nr)	unifiziert value mit dem ersten Term, der unter key mit db_ref_nr liegt.
erase(+db_ref_nr)	löscht den Term oder die Klausel