Einführung in das Symbolische Programmieren (Prolog)

Hans Leiß

CIS, LMU München, WS 2011/12

31. Januar 2012

Organsiatorisches

1. Vorlesung: Di 16-18, Oettingenstr.67, B 001
2. Klausur (VL): Di 7.2.2012, 16-18, Oe67, B 001
3. Tafelübung: Do 16-18, Oettingenstr.67, Raum 157
4. Klausur (TÜ): Do 9.2.2012, 16-18, Oe67, Raum 157
5. Abmeldephase: 24.10.-11.12.2011
6. Sprechstunde: Mi 13-15 Uhr, Oettingenstr.67, C 110
7. Kursseite:
   `http://www.cis.uni-muenchen.de/~leiss/symb.Progr.-11/`
8. Tutorin: Katharina Groth
   Email: `katharina-groth@web.de`
   Fragestunde: Di 18-20 im Rechnerraum BU102 (Kalahari), Oettingenstr.67

Programmiersprache: SWI-Prolog

Woher bekommt man (kostenlos) SWI-Prolog?

- SWI-Prolog/XPCE
  `http://www.sci-prolog.org/download/stable`
- Stabile Version: 5.10.5(4) für MS-Windows, Linux, MacOS X
- Linux-Distributionen für Debian, Ubuntu, Redhat, SuSe
- Programmname `bin/swipl` oder `bin/xpce`

Programme editieren:

- in jedem Editor
- in dem in SWI-Prolog eingebauten (Mini-)Emacs
- im normalen Emacs-Editor (mit Formatierungs- und Ladehilfe)

Symbolisches Programmieren

- Nicht-numerisches Programmieren
- Manipulieren von „symbolischen Ausdrücken“
- Logisches Programmieren:
  - Programm definert logische Prädikate
  - Anwendung: suche Objekte, die die Prädikate erfüllen

Künstliche Einteilung:

- aussagenlogisches Programmieren (PROPLOG)
- logisches Programmieren mit einfachen Daten (DATALOG)
- reines Prolog (ohne Seiteneffekte) (Pure PROLOG)
- volles Prolog (mit Seiteneffekten) (PROLOG)

PROPLOG – Aussagenlogisches Prolog

Syntax der Programme

```
Program      :=  Clauses
Clauses      :=        (leeres Programm)
              |  Clause Clauses

Clause       :=  PropSymbol.
              |  PropSymbol :- Formula.

PropSymbol   :=  [a-z]([a-zA-Z0-9_])*

Formula      :=  PropSymbol
              |  PropSymbol, Formula

Goal         :=  Formula.
```

Bedeutung von Klauseln

Ein *Fakt* ist eine Klausel der Form

$$\varphi.$$

Sie bedeutet, daß die Aussage φ als wahr angenommen wird.

Eine *Regel* ist eine Klausel der Form

$$\varphi \texttt{ :- } \varphi_1, \ldots, \varphi_n.$$

Sie bedeutet, daß φ als wahr gilt, wenn φ_1 und ... und φ_n als wahr gelten.

Aussagenlogische Konstante

Die speziellen PropSymbole `true`,`fail` haben die feste Bedeutung:

`true`	gilt als wahr (bewiesen),
`fail`	gilt als nicht wahr (unbeweisbar).

Deren Bedeutung kann man nicht ändern! Zum Beispiel wird die Regel

```
fail :- true.
```

als unzulässig vom Prolog-Compiler abgelehnt.

Es gibt eine ganze Reihe solcher *eingebauter Konstanten* in Prolog, deren Bedeutung der Programmierer nicht ändern kann.

Bedeutung von Zielen (Goals)

Ein *Ziel* (Goal)

$$\texttt{?-}\ \varphi_1, \ldots, \varphi_n.$$

ist eine *Anfrage*: Sind, wenn man die Klauseln des Programms annimmt, die Aussagen φ_1 und ... und φ_n wahr?

Die *Bedeutung einer Frage* ist die Menge der korrekten Antworten. Als Antwort auf die Frage ?- φ kann man zweierlei in Betracht ziehen:

(i) den Wahrheitswert von φ, oder

(ii) eine Begründung (bzw. Widerlegung) von φ.

PROLOG antwortet im Sinne von (ii).

Der PROPLOG-Interpreter

Um bei gegebenem Programm eine Anfrage zu beantworten, wird die Anfrage als ein Beweisziel verstanden und ein Beweis dafür gesucht.

Die Idee der Suche nach einem Beweis für eine Liste von Zielen ist:

1. falls die Liste der Ziele leer ist, antworte mit `Yes`;
2. sonst nimm das erste Ziel und suche im Programm die nächste Regel, deren Kopf gleich dem Ziel ist;
3. wenn die Regel den Rumpf `true` hat, entferne das Ziel aus der Liste, andernfalls ersetze es durch den Rumpf der Regel; nun beweise die neue Liste von Zielen;
4. falls es in 2. keine zum Ziel passende Regel gibt, gehe zum vorherigen Schritt 2. zurück und versuche es mit der nächsten passenden Regel;
5. falls es keine weitere Möglichkeit gibt, antworte mit `No`.

Beweissuche in PROPLOG als Pseudoprogramm

Gegeben sei ein PROPLOG-Programm `P = [p1,...,pn]`.

⟨*Pseudoprogramm zur Beweissuche*⟩≡

```
provable(goal) =
    begin
     if goal=[] then return(Yes)
     else
          for i=1 to n do
          {if head(phi.i) = first(goal) then
              if provable(append(body(phi.i),
                                  rest(goal))) then
                 return(Yes)
              end
         end};
         return(No)
    end
```

Das Pseudoprogramm zur Beweissuche benutzt folgende

⟨*Hilfsfunktionen zur Beweissuche*⟩≡

```
head(p :- q1,...,qn.) = p                    head(p.) = p
body(p :- q1,...,qn.) = [q1,...,qn]          body(p.) = []

append([p1,...,pk],[q1,...qn]) = [p1,...,pk,q1,...,qn]

first([q1,...,qn]) = q1
rest([q1,...,qn]) = [q2,...,qn]
```

Sei $[\varphi_1, \ldots, \varphi_n]$ die Liste der Klauseln von P in der Anordnung, wie sie in P erscheinen, und $[\psi_1, \ldots, \psi_n]$ die Liste der Aussagen (PropSymbole) des Ziels ?- ψ.

Der Suchbaum zum Programm P und Ziel ψ

Einen Überblick darüber, wie Prolog (bzw. PROPLOG) das Ziel ψ bei gegebenem Programm P abarbeitet, gibt der *Suchbaum* zu P und ψ, den man wie folgt erhält:

1. An der Wurzel steht die Liste der Ziele aus ψ.
2. Steht an einem Knoten im Baum die Liste `[p,q,...]` und hat das Programm k Regeln mit Kopf `p`, so verzweigt der Knoten zu k Teilbäumen;
3. wenn die i-te der k Regeln die Regel `p :- r,...,s` ist, so steht an der Wurzel des i-ten Teilbaums die Liste `[r,...,s,q,...]`.

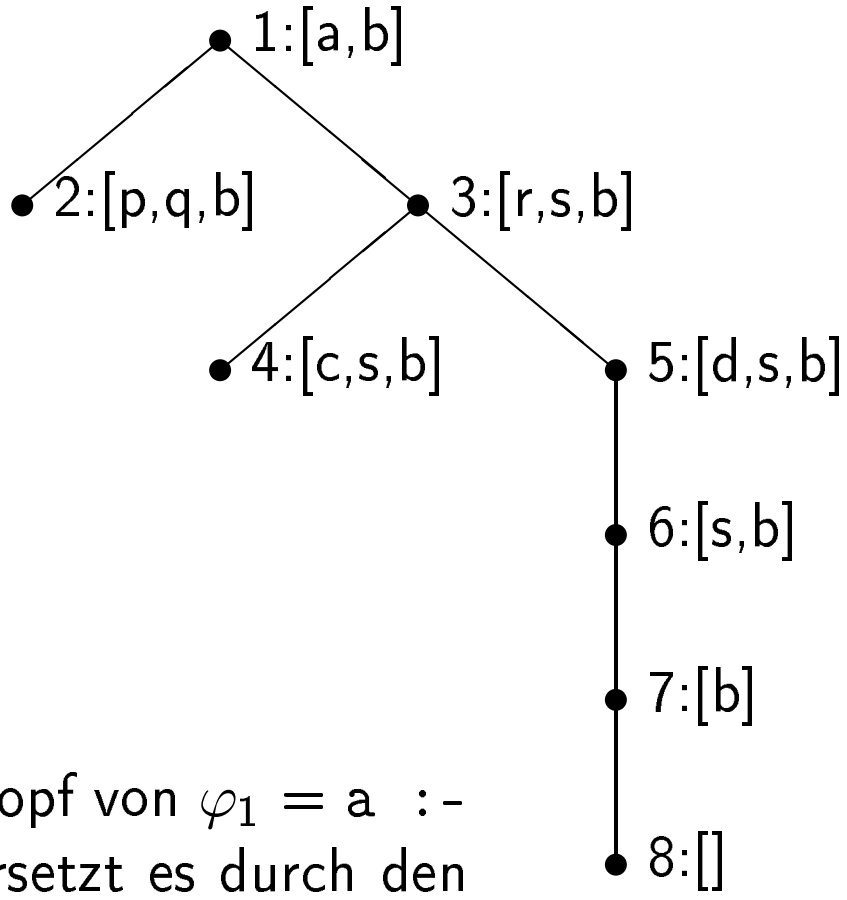
Gibt es zum ersten Ziel in der Liste eines Knotens keine passende Programmregel, so ist der Knoten ein Blatt des Baums.
Der Baum wird mit *Tiefensuche* durchlaufen: zuerst wird die Wurzel besucht, dann wird (auf die gleiche Weise) der linkeste Teilbaum durchlaufen, dann der zweite Teilbaum usw., bis zum Blatt `[]`.

Beweissuche und Suchbaum am Beispiel

Programm:

φ_1	`a :- p, q.`
φ_2	`a :- r, s.`
φ_3	`b.`
φ_4	`d.`
φ_5	`r :- c.`
φ_6	`r :- d.`
φ_7	`s.`

Anfrage: `?- a,b.`



Beim Ziel `1:[a,b]` paßt der Regelkopf von φ_1 = `a :- p,q.` zum ersten Teilziel `a`; man ersetzt es durch den Rumpf `p,q` und kommt zum neuen Ziel `2:[p,q,b]`. Da kein Regelkopf auf das Teilziel `p` paßt, muß man *zurücksetzen* und beim Ziel `1` eine andere passende Regel versuchen, hier φ_2 = `a:- r,s.`

Scheiternde Beweissuche

Nicht auf jede Anfrage findet die Beweissuche *provable* eine Antwort.

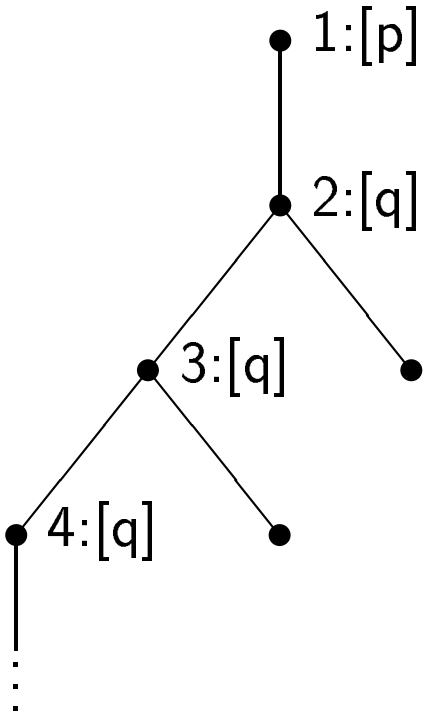
Betrachte die Anfrage `?- p.` an die Programme

```
p :- q.                 p :- q.
q :- q.      bzw.       q.
q.                      q :- q.
```

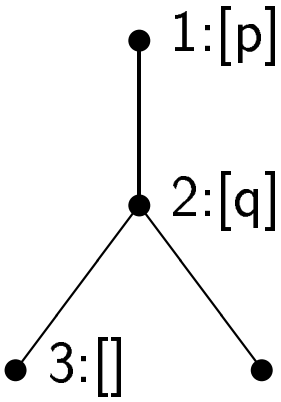
Man erhält links keine Antwort: Um das Teilziel `q` zu beweisen, wird die erste Regel `q:- q.` benutzt, dann muß deren Rumpf `q` bewiesen werden. *Erst wenn man dabei alle Möglichkeiten versucht hat*, wird die zweite zu `q` passende Klausel, `q.`, versucht. Diese erreicht man nicht.

Beim Programm rechts wird das Teilziel `q` durch die erste passende Klausel, `q.`, bewiesen, und man erhält das Ziel `3:[]`.

Der Suchbaum ist



bzw.



Die Reihenfolge der Programmregeln beeinflußt also das Ergebnis!

Negation bzw. Unbeweisbarkeit

PROPLOG und Prolog haben keine (logische) Negation:
`fail` (bzw. No) bedeutet nicht das Gegenteil von `true` (bzw. Yes).

Bei dem Programm

⟨*Beispiel* ⟩≡

```
p :- q, r.
q.
```

erhält man auf die Anfrage `?- r.` die Antwort `fail`.

Die Antwort `No` auf die Frage `?-` φ besagt nicht, daß $\neg\varphi$ aus P beweisbar ist, sondern nur, *daß Suche nach einem Beweis für* φ *terminiert und scheitert*, d.h. alle Möglichkeiten sind erschöpft, ohne daß ein Beweis von φ aus P gefunden wurde.

Es braucht deshalb aber noch keinen Beweis für $\neg\varphi$ zu geben.

Es gibt also drei mögliche Antworten auf ?- φ., nämlich

`Yes`,	falls φ aus P beweisbar ist,
`No`,	falls die Unbeweisbarkeit von φ aus P in endlich vielen Schritten feststellbar ist, (finite failure)
divergiert,	sonst.

Man kann PROPLOG (wie Prolog) um ein *not* erweitern durch:

$$\begin{aligned} \mathit{provable}(\mathit{not}(\varphi)) = \texttt{Yes} \quad &:\Longleftrightarrow \quad \mathit{provable}(\varphi) = \texttt{No}, \\ \mathit{provable}(\mathit{not}(\varphi)) = \texttt{No} \quad &:\Longleftrightarrow \quad \mathit{provable}(\varphi) = \texttt{Yes}. \end{aligned}$$

Man nennt *not* die *negation as failure*.
Offizielle Notation: `\+(p)` statt `not(p)`, für "`p` ist unbeweisbar".

Konjunktion und Disjunktion

Die *Konjunktion* ($\varphi_1 \wedge \varphi_2$) zweier Aussagen φ_1 und φ_2 ist durch die Programmsyntax unter

$$\begin{array}{lcl} \text{Formula} & := & \text{PropSymbol} \\ & \mid & \text{PropSymbol, Formula} \end{array}$$

erlaubt, und zwar implizit wie folgt geklammert:

`p, q, r` für $(p \wedge (q \wedge r))$

Die *Disjunktion* ($\varphi_1 \vee \varphi_2$) zweier Aussagen ist unter *Formula* nicht enthalten. Man kann sie aber wie folgt mit einer neuen Aussagevariablen simulieren:

```
s :- p.
s :- q.
s :- r.
```

für $(p \vee (q \vee r))$

PROLOG erlaubt `(p,q)`, `(p;q)`, `not(p)` bzw. `\+(p)` als Aussagen:

$$
\begin{array}{lcl}
\text{Formula} & := & \text{PropSymbol} \\
 & | & \text{(Formula, Formula)} \\
 & | & \text{(Formula; Formula)} \\
 & | & \text{not(Formula).}
\end{array}
$$

entsprechend der Syntax der Aussagenlogik:

$$
\begin{array}{lcll}
\varphi, \psi & := & p & \\
 & | & (\varphi \wedge \psi) & \text{„und“} \\
 & | & (\varphi \vee \psi) & \text{„oder“} \\
 & | & \neg\varphi & \text{„nicht“}
\end{array}
$$

Klammerregel:
Wer $((p \vee q) \wedge r)$ meint, muß auch `((p;q),r)` oder `(p;q),r` schreiben. `p;q,r` wird als `p;(q,r)` gelesen!

Beachte: Die Bedeutung von `(p,q)` bzw. `(p;q)` sind durch die Beweissuche definiert, i.a. verschieden von `(q,p)` bzw. `(q;p)`.

Anwendungsprogramm Prüfungsvoraussetzung

⟨*Regeln zur Zulassung zur CL-Zwischenprüfung*⟩≡

```
erfuellePruefungsvoraussetzungenHCL :-
        habeCL1Schein,    % alle Scheine erforderlich
        habeCL2Schein,
        habeLinguistikSchein,
        habeMatheSchein.
habeLinguistikSchein :-   % ein Schein erforderlich
        habeSyntaxSchein.
habeLinguistikSchein :-
        habeMorphologieSchein.
habeLinguistikSchein :-
        habeSemantikSchein.
habeMatheSchein :-        % ein Schein erforderlich
        habeMathe1Schein; habeMathe2Schein;
        habeAutomatenSchein; habeStatistikSchein.
```

⟨*Fakten bei einem Studenten im 2.Semester (anzupassen)* ⟩≡

```
habeCL1Schein :- true.
habeCL2Schein :- fail.
habeSyntaxSchein :- fail.
habeMorphologieSchein :- fail.
habeSemantikSchein :- fail.
habeMathe1Schein :- true.
habeMathe2Schein :- fail.
habeAutomatenSchein :- fail.
habeStatistikSchein :- fail.
```

⟨*Anfrage nach jedem Semester:*⟩≡

```
?- erfuellePruefungsvoraussetzungenHCL.
```

DATALOG - Prolog mit atomaren Individuen

Syntax der Programme

Program	:=	Clauses
Clauses	:=	
	|	Clause Clauses
Clause	:=	Atom.
	|	Atom :- Atoms.
Atoms	:=	Atom
	|	Atom, Atoms
Atom	:=	RelSymbol
	|	RelSymbol(Terms)

Goal	:=	Atoms.
Terms	:=	Term
	|	Term, Terms
Term	:=	VarSymbol
	|	ConstSymbol

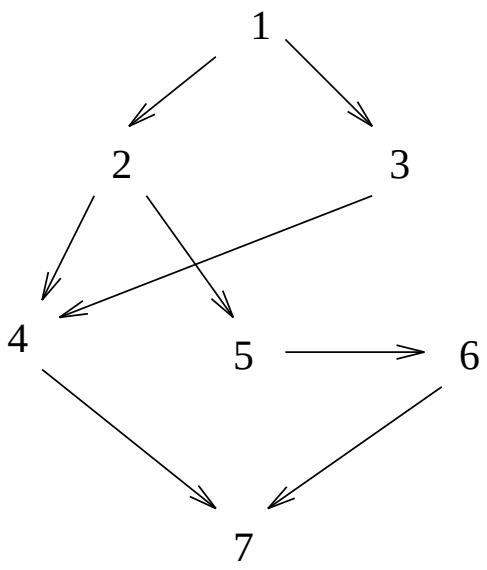
RelSymbol	:=	[a-z]([a-zA-Z0-9_])*
ConstSymbol	:=	$[0\text{-}9]^+$
	|	[a-z]([a-zA-Z0-9_])*
VarSymbol	:=	[A-Z]([a-zA-Z0-9_])*
	|	_([a-zA-Z0-9_])*

Wege in kreisfreien Graphen

Gegeben sei ein kreisfreier Graph $\mathcal{G} = (\{1, \ldots, m\}, \mathit{kante})$:

⟨*Kanten im Graph*⟩≡

```
kante(1,2).
kante(1,3).
kante(2,4).
kante(2,5).
kante(3,4).
kante(4,7).
kante(5,6).
kante(6,7).
```



Definiere, wann es in Graphen einen Weg von X nach Y gibt:

⟨*Definition von Wegen*⟩≡

```
weg(X,Y) :- kante(X,Y).
weg(X,Y) :- kante(X,Z), weg(Z,Y).
```

Wir wollen an dieses Programm Fragen stellen wie

⟨*Prologanfragen*⟩≡

```
?- weg(1,6).
?- weg(3,7).
?- weg(X,4).
```

Die letzte Frage soll bedeuten: von welchem Punkt X gibt es einen Weg nach Punkt 4? (mehrere Antworten!)

Logische Bedeutung der zweiten Regel:

$$\forall X\, \forall Y\, \forall Z\, (kante(X, Z) \wedge weg(Z, Y) \rightarrow weg(X, Y)).$$

Logisch äquivalent dazu ist:

$$\forall X\, \forall Y\, (\exists Z\, (kante(X, Z) \wedge weg(Z, Y)) \rightarrow weg(X, Y)).$$

Intuitive Beschreibung der DATALOG-Beweissuche

Sei $P = [\varphi_1, \ldots, \varphi_n]$ ein Program und $G = [\psi, \chi, \ldots]$ die Liste der (noch) zu beweisenden Teilziele.

1. Wähle aus G das nächste Teilziel, ψ.
2. Durchsuche P nach der ersten Klausel φ_i, die zu ψ „paßt“, d.h. bilde φ_i' durch Umbenennen der Variablen von φ_i und suche eine Substitution S, so daß $\mathit{Kopf}(\varphi_i')S = \psi S$. Das geht mit $S := \mathtt{match}(\mathit{Kopf}(\varphi_i'), \psi)$, wenn überhaupt.
3. Durch 2. wurde ψS mit

   $$\mathit{Kopf}(\varphi_i')S \ \text{:-}\ \mathit{Körper}(\varphi_i')S$$

   bewiesen, das erste Teilziel von $GS = [\psi S,\ \chi S, \ldots]$.
4. Suche einen Beweis für die neuen Ziele

   $$G' := [\mathit{Körper}(\varphi_i')S,\ \chi S,\ \ldots].$$

Wenn ein Beweis gefunden wurde, ist die *Antwort* die Belegung S, *soweit sie die Variablen der Anfrage G betrifft*.

Match(φ, ψ) zweier Atome

1. vergleiche φ mit ψ nach ihrer Form, und
2. belege dabei –bei gleichem Relationszeichen–
   - 2.1 Variable in φ durch (spezialisierte) Teilterme der entsprechenden Stellen von ψ,
   - 2.2 Variable in ψ durch (spezialisierte) Teilterme der entsprechenden Stellen von φ),

   (erst linke, dann rechte Argumente vergleichen; die Variablen auch in anderen Argumenten belegen).

Beweissuche in DATALOG als Pseudoprogramm

Sei P durch die Liste $[\varphi_1, \ldots, \varphi_n]$ von Klauseln und die Anfrage ψ durch die Liste $G = [\psi_1, \ldots, \psi_k]$ von Atomen gegeben.

Bei der Antwortsuche auf die Frage ?- ψ. soll der Vergleich $\mathit{head}(\varphi_i) = \mathit{first}(G)$ abgeschwächt werden: Variable X dürfen durch (atomare) Terme t belegt werden. (vgl. *match* unten)

Jedesmal, wenn eine Klausel benutzt wird, wird eine Kopie der Klausel gemacht, in der die Variablen durch „neue“ Variablen ersetzt sind:

⟨*Hilfsfunktion*⟩≡

```
rename(p(X,Y) :- q(X,Z),r(Y,Z).)
     :=  p(Xj,Yj) :- q(Xj,Zj),r(Yj,Zj). (*neue Vars*)
```

⟨*Beweissuche für G*⟩≡

```
function provable(G)=
 begin
   if G = [] then return(Yes)
   else
      for i=1 to n do
        {C := rename(phi_i);
         S := match(head(C),first(G));
         if S =/= No then
            G' := apply(S,append(body(C),rest(G)));
            if provable(G') then
               return(Yes)
            end
         end};
      return(No)
   end
 end
```

Bleibt zu erklären: S, $match(\psi, \psi')$ $apply(S, [\psi_1, \ldots, \psi_k])$.

Hilfsbegriffe und -funktionen

Eine *Substitution* (oder *Belegung*) S ist eine endliche Menge

$$\{t_1/X_1, \ldots, t_n/X_n\}$$

von *Bindungen* t_i/X_i von Termen t_i an Variable X_i.

Die *Anwendung* der Substitution S auf ein Atom ψ, $apply(S, \psi)$ oder kurz ψS, ist definiert durch

$$\begin{array}{lcl} rS & := & r \\ r(s_1, \ldots, s_n)S & := & r(s_1 S, \ldots, s_n S) \\ \\ cS & := & c \\ XS & := & \begin{cases} t, & \text{falls } t/X \in S, \\ X, & \text{sonst.} \end{cases} \end{array}$$

und die Anwendung von S auf eine Liste von Atomen durch

$$[\psi_1, \ldots, \psi_n]S \quad := \quad [\psi_1 S, \ldots, \psi_n S].$$

Beispiel:

$$
\begin{aligned}
&[s(X,Y),\ r(X,c)]\{Y/X, d/Y\} \\
&\quad = [s(X,Y)\{Y/X,d/Y\},\ r(X,c)\{Y/X,d/Y\}] \\
&\quad = [s(X\{Y/X,d/Y\}, Y\{Y/X,d/Y\}),\ \ldots] \\
&\quad = [s(Y,d),\ r(Y,c)].
\end{aligned}
$$

Die Variablen werden simultan ersetzt, nicht nacheinander.

Auf einander folgende Substitutionen bilden eine neue:

$$
\begin{aligned}
&compose(\{t_1/X_1, ..., t_n/X_n\}, T) := \\
&\quad \{t_1 T/X_1, ..., t_n T/X_n\} \cup \{t/X \mid t/X \in T, X \notin \{X_1, \ldots, X_n\}\}
\end{aligned}
$$

Beispiel:

$$
\begin{aligned}
compose(\{Y/X, 3/Z\}, \{d/Y\}) &= \{Y\{d/Y\}/X, 3\{d/Y\}/Z\} \\
&= \{d/X, 3/Z\}
\end{aligned}
$$

⟨*function match(A:Atom,B:Atom)*⟩≡

```
begin
  if (relsymbol(A) =/= relsymbol(B)
      or  arity(A) =/= arity(B))
     then return(No)
  else
    S := {};
    for i:= 1,...,arity(A) do
      {if variable(arg(A,i)) then
          S := compose(S,{arg(B,i)/arg(A,i)});
          A := apply(S,A); B := apply(S,B)
       elseif variable(arg(B,i)) then
          S := compose(S,{arg(A,i)/arg(B,i)});
          A := apply(S,A); B := apply(S,B)
       elseif  arg(A,i) =/= arg(B,i)
          then return(No)
       end};
    return(S)
  end
```

⟨*Hilfsfunktionen für match*⟩≡

```
relsymbol(r(t1,...,tn))  :=  r  =: relsymbol(r)

arg(r(t1,..,tn),i) := ti,    falls 0 < i < n+1

variable(X) := true,         variable(c) := false
```

Beispiel zum Matchen

Wir berechnen

$$match(A, B)$$

für die Atome

$$A := r(X, a, X) \qquad \text{und} \qquad B := r(Y, Y, b).$$

Da beide Atome dasselbe 3-stellige Relationssymbol r haben, beginnt man mit der leeren Substitution

$$S := \{\}$$

und vergleicht der Reihe nach die Argumente $i = 1, 2, 3$:

$$A := r(X, a, X) \qquad B := r(Y, Y, b)$$

$i = 1$: Das erste Argument von A ist eine Variable X und wird mit Y, dem ersten Argument von B, belegt:

$$\begin{array}{lcllcl} S & := & \{\}\{Y/X\} & = & \{Y/X\}; \\ A & := & r(X, a, X)\{Y/X\} & = & r(Y, a, Y); \\ B & := & r(Y, Y, b)\{Y/X\} & = & r(Y, Y, b); \end{array}$$

$i = 2$: Das zweite Argument von A ist eine Konstante a, mit der Y, das zweite Argument von B belegt wird:

$$\begin{array}{lcllcl} S & := & \{Y/X\}\{a/Y\} & = & \{a/X, a/Y\}; \\ A & := & r(Y, a, Y)\{a/X, a/Y\} & = & r(a, a, a); \\ B & := & r(Y, Y, b)\{a/X, a/Y\} & = & r(a, a, b); \end{array}$$

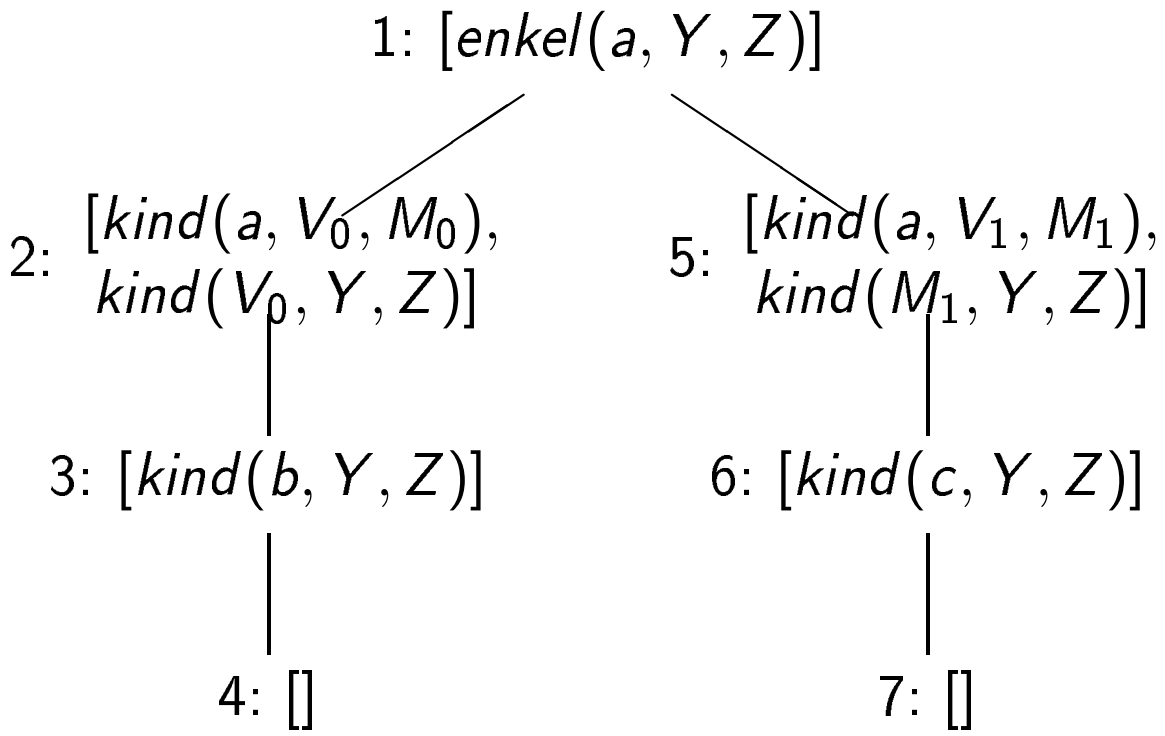
$i = 3$: Das dritte Argument von A ist die Konstante a, das dritte Argument von B die Konstante b. Da diese Konstanten verschieden sind, ist das Ergebnis $S := No$.

Beispiel einer DATALOG-Anfrage

Das Programm $P = [\varphi_1, \ldots, \varphi_6]$ sei:

```
enkel(X,GV,GM) :- kind(X,V,M), kind(V,GV,GM).
enkel(X,GV,GM) :- kind(X,V,M), kind(M,GV,GM).

kind(a,b,c). kind(b,d,e). kind(c,f,g). kind(d,h,i).
```

Der Suchbaum (s.Folie unten) zur Frage `?- enkel(a,Y,Z).` ist:



Für die linke Lösung (d.h. `enkel(a,d,e)`) wird benutzt:

$$
\begin{aligned}
&\mathit{match}(\mathit{enkel}(X_0, GV_0, GM_0), \mathit{enkel}(a, Y, Z)) = \\
&\qquad\qquad\qquad\qquad \{a/X_0, Y/GV_0, Z/GM_0\}, \\
&\mathit{match}(\mathit{kind}(a, b, c), \mathit{kind}(a, V_0, M_0)) = \{b/V_0, c/M_0\}, \\
&\mathit{match}(\mathit{kind}(b, d, e), \mathit{kind}(b, Y, Z)) = \{d/Y, e/Z\}.
\end{aligned}
$$

Der Suchbaum zu einer Frage

Der Suchbaum zu einer Liste G von (atomaren) Zielen bei einem Programm P wird wie folgt gebildet:

1. Die Wurzel des Baums ist die Liste G von Zielen.
2. Stehen am Knoten K des Baums die Ziele $[G_1, G_2, ...]$ und hat G_1 das n-stellige Prädikatszeichen r und hat P m Klauseln für dieses r, etwa

   $$(i_1)\ r(\vec{t}_1) :- Rumpf_1. \quad \ldots \quad (i_m)\ r(\vec{t}_m) :- Rumpf_m.,$$

   so verzweigt der Baum bei K zu m Nachfolgerknoten $K_1, \ldots, K_m$ (den Wurzeln von m Teilbäumen).
   An den Zweig von K zu K_j schreibt man die Nummer $j \in \{i_1, \ldots, i_m\}$ der zu benutzenden Klausel und die Belegung S_j, die G_1 mit dem Kopf $r(\vec{t}_j)$ der Klausel (j) (bzw. ihrer Kopie mit frischen Variablen) gleich macht.
   Falls es keine solche Belegung S_j gibt, streichen wir den Zweig durch oder schreiben `fail` darunter.[1]

[1] Aus Platzgründen werden solche Zweige oft weggelassen. Leider erlaubt mein Graphikprogramm nicht, die Kanten zu beschriften.

3. Gibt es S_j, so schreibt man an den Knoten K_j die Ziele

$$[Rumpf_j S_j, G_2 S_j, \ldots],$$

   die aus denen bei K entstehen, indem man G_1 durch $Rumpf_j$ ersetzt (bzw. G_1 wegläßt, wenn $Rumpf_j = \texttt{true}$ ist) und die Belegung S_j darauf anwendet.
4. Dann macht man dasselbe mit $K_1, \ldots, K_m$ und ihren Zielen, bis man nur noch Knoten mit leeren Ziellisten bekommt. (Der Baum kann i.a. unendlich tief werden!)

PURE PROLOG - Prolog mit komplexen Individuen

Syntax der Programme

Program	:=	Clauses	Goal	:=	Atoms.
Clauses	:=				
	|	Clause Clauses	Terms	:=	Term
Clause	:=	Atom.		|	Term, Terms
	|	Atom :- Atoms.	Term	:=	VarSymbol
Atoms	:=	Atom		|	ConstSymbol
	|	Atom, Atoms		|	FunSymbol(Terms)
Atom	:=	RelSymbol			
	|	RelSymbol(Terms)			

RelSymbol	:=	[a-z]([a-zA-Z0-9_])*
FunSymbol	:=	RelSymbol
ConstSymbol	:=	$[0\text{-}9]^+$
	|	[a-z]([a-zA-Z0-9_])*
VarSymbol	:=	[A-Z]([a-zA-Z0-9_])*
	|	_([a-zA-Z0-9_])*

Beispiel: Arithmetik

Wir können die natürlichen Zahlen als Terme kodieren, durch

- ein Prädikat `nat/1` zur Aussonderung der Terme, die natürliche Zahlen darstellen,
- eine Konstante 0 für die kleinste natürliche Zahl,
- ein Funktionszeichen s für den Nachfolger $n \mapsto n + 1$.

Damit definieren wir die Zahlen durch folgendes Programm:

⟨*Zahlendefinition*⟩≡

```
nat(0).
nat(s(X)) :- nat(X).
```

Die Zahl n wird durch den Term $\overbrace{s(s(\ldots s}^{n\ mal}(0)\ldots))$ kodiert.

Prädikate auf natürlichen Zahlen

Nun können wir die Addition natürlicher Zahlen als Relation zwischen entsprechenden Termen definieren:

⟨*Addition*⟩≡

```
add(X,0,X) :- nat(X).
add(X,s(Y),s(Z)) :- add(X,Y,Z).
```

Die Anordnung wird durch die Relation `leq/2` definiert:

⟨*Anordung*⟩≡

```
leq(0,X) :- nat(X).
leq(s(X),s(Y)) :- leq(X,Y).
```

und damit das Maximum zweier natürlicher Zahlen durch:

⟨*Maximum*⟩≡

```
max(0,Y,Y) :- nat(Y).
max(s(X),0,s(X)) :- nat(X).
max(s(X),s(Y),s(Z)) :- max(X,Y,Z).
```

Beweissuche für PURE PROLOG

An der Beweissuche ändert sich nichts außer dem Vergleich von atomaren Formeln. Um komplexe Terme behandeln zu können, ändern wir den Vergleich von Atomen zu

⟨*match(A:Atom,B:Atom)*⟩≡

```
function match(A:Atom,B:Atom) =
 begin
   if (relsymbol(A) =/= relsymbol(B)
       or  arity(A) =/= arity(B))
      then return(No)
   else
     S := {};
     for i:= 1,...,arity(A) do {
        T := unify(arg(A,i),arg(B,i));
        if T =/= No
        then
           S := compose(S,T);
           A := apply(S,A); B := apply(S,B)
        else return(No) };
     return(S)
   end
 end
```

Unifizieren komplexer Terme: Unify(s, t)

Wir suchen –wie beim Vergleichen von Atomen– eine Substitution S, die die beteiligten Terme „gleich macht“.

Da geht ähnlich wie beim Matchen in DATALOG, aber die Teilterme müssen berücksichtigt werden:

1. vergleiche die Funktionszeichen von s und t, und –bei gleichem Funktionszeichen– vergleiche der Reihe nach die Argumentterme von s mit denen von t,
2. belege*) dabei überall
   - 2.1 Variable in Teiltermen von s durch (spezialisierte) Teilterme der entsprechenden Stellen von t,
   - 2.2 Variable in Teiltermen von t durch (spezialisierte) Teilterme der entsprechenden Stellen von s,

*) <u>falls</u> folgende Einschränkung erfüllt ist, der sogenannte

Occurs check: X darf nicht durch einen komplexen Term belegt werden, in dem X vorkommt!

In DATALOG sind alle Terme t Variable oder Konstante, und man hat immer

$$match(X, t) = \{t/X\}, \qquad \text{da } X\{t/X\} = t = t\{t/X\}.$$

In PURE PROLOG haben wir auch *komplexe* Terme. Für den Term $t := f(X)$ z.B. muß aber gelten:

$$unify(X, t) = No, \qquad \begin{array}{l} \text{denn für } \textit{jedes } s \text{ ist } X\{s/X\} = s, \\ \text{aber } t\{s/X\} = f(X)\{s/X\} = f(s) \neq s. \end{array}$$

Beispiel zum Unifizieren

Wir berechnen

$$unify(s, t)$$

für die Terme

$$s := f(X, g(Y)) \qquad \text{und} \qquad t := f(h(Y), X).$$

Da beide Terme dasselbe Funktionssymbol f haben, beginnt man mit der leeren Substitution

$$T := \{\}$$

und vergleicht der Reihe nach die Argumente $i = 1, 2$:

$i = 1$: Das erste Argument von s ist eine Variable X und wird mit $h(Y)$, dem ersten Argument von t, belegt:

$$\begin{array}{lcllcl} T & := & \{\}\{h(Y)/X\} & = & \{h(Y)/X\}; \\ s & := & f(X, g(Y))\{h(Y)/X\} & = & f(h(Y), g(Y)); \\ t & := & f(h(Y), X)\{h(Y)/X\} & = & f(h(Y), h(Y)); \end{array}$$

$i = 2$: Das zweite Argument von s ist jetzt der Term $g(Y)$, das zweite Argument von t der Term $h(Y)$.
Dies sind komplexe Terme mit *verschiedenen* Funktionszeichen, also ist die Antwort

$$T := No.$$

⟨*Unify(A:Term,B:Term), einfache Fälle*⟩≡

```
function unify(A:Term,B:Term) =
 begin
   if variable(A) then
      if (A does not occur in B)
      then return({B/A})
      else if B = A then return({})
                    else return(No)

   elseif variable(B) then
      if (B does not occur in A)
      then return({A/B}) else return(No)

   elseif (funsymbol(A) =/= funsymbol(B)
           or  arity(A) =/= arity(B)) then
      return(No)
```

⟨*Unify(A:Term,B:Term), sonst-Fall*⟩≡

```
    else
       S := {};
       for i:= 1,..., arity(A) do
         {
          Si := unify(arg(A,i),arg(B,i));
          if Si =/= No
          then S := compose(S,Si);
               A := apply(S,A);
               B := apply(S,B)
          else return(No)
         };
       return(S)
  end
```

```
funsymbol(f(t1,...,tn))  :=  f  =: funsymbol(f)
arity(f(t1,...,tn)) := n
```

Beispiel Addition: Rekursive Berechnung

Betrachte zum obigen Programm P das Ziel

$$?\text{-}\ add(s(s(0)), s(0), Z).$$

Es wird wie folgt abgearbeitet:

$$1 : [add(s(s(0)), s(0), Z)]$$

$$\Big|$$

$$2 : [add(s(s(0)), 0, Z')]$$

$$\Big|$$

$$3 : [nat(s(s(0)))]$$

$$\Big|$$

$$4 : [nat(s(0))]$$

$$\Big|$$

$$5 : [nat(0)]$$

Beispiel Addition (Forts.)

Im ersten Schritt benutzt man die zweite *add*-Klausel und

$$\begin{aligned} & \mathit{unify}(\mathit{add}(s(s(0)), s(0), Z), \mathit{add}(X', s(Y'), s(Z'))) \\ = \quad & \{s(s(0))/X', 0/Y', s(Z')/Z\} =: S_1. \end{aligned}$$

Im zweiten Schritt benutzt man die erste *add*-Klausel und

$$\begin{aligned} & \mathit{unify}(\mathit{add}(s(s(0)), 0, Z'), \mathit{add}(X'', 0, X'')) \\ = \quad & \{s(s(0))/X''\}\ \mathit{unify}(Z'\{s(s(0))/X''\}, X''\{s(s(0))/X''\}) \\ = \quad & \{s(s(0))/X''\}\ \mathit{unify}(Z', s(s(0))) \\ = \quad & \{s(s(0))/X''\}\ \{s(s(0))/Z'\} \\ = \quad & \{s(s(0))/X'',\ s(s(0))/Z'\} =: S_2. \end{aligned}$$

Die Belegung der Zielvariablen Z durch $S_1 S_2$ ist

$$\begin{aligned} Z\ S_1\ S_2 &= s(Z')\ S_2 \\ &= s(Z' S_2) \\ &= s(s(s(0))). \end{aligned}$$

Also ist das Ergebnis ein Beweis von

$$add(s(s(0)), s(0), s(s(s(0)))),$$

was der Aussage $2 + 1 = 3$ entspricht.

Zur Terminologie match/unify

Für DATALOG wurde

$$\mathit{match}(s{:}\mathit{Term},\ t{:}\mathit{Term})$$

für den einfachen Fall von

$$\mathit{unify}(s{:}\mathit{Term},\ t{:}\mathit{Term})$$

verwendet, bei dem s und t atomar sind: in diesem Fall braucht man keinen *occurs check*.

Im allgemeinen versteht man aber unter *Matching* etwas anderes als unter *Unifikation*: Eine Substitution S ist

1. ein *Matcher* von r und t, falls $r = tS$ ist, und
2. ein *Unifikator* von r und t, falls $rS = tS$ ist.

Der Term r *passt in* das Muster t (r *matches* t), falls es einen Matcher von r und t gibt.

Beispiel: $f(g(Y), 3)$ passt in $f(X, Y)$, da

$$f(g(Y), 3) = f(X, Y)\{g(Y)/X, 3/Y\}.$$

Die Terme r und t sind *unifizierbar*, wenn es einen Unifkator von r und t gibt.

Beispiel: $f(g(Y), Z)$ und $f(g(3), Y)$ sind unifizierbar, da

$$f(g(Y), Z)\{3/Y, 3/Z\} = f(g(3), Y)\{3/Y, 3/Z\}.$$

Dagegen sind $f(g(Y), 4)$ und $f(g(3), Y)$ nicht unifizierbar: ein Unifikator S müsste erfüllen:

$$YS = 3 \quad \text{und} \quad 4 = YS.$$

Listen in Prolog

Ein eingebauter 'Datentyp' von Prolog sind die *Listen*. Das sind Terme, die in spezieller Weise aufgebaut sind mit

1. der Konstanten `[]` für die *leere Liste*, und
2. dem 2-stelligen Funktionszeichen '.' für die *Paarbildung*. Statt `'.'(s,t)` kann man `[s|t]` schreiben.

Die *Listen* sind die Terme t, für die *liste*(t) gilt, wobei das Prädikat `liste/1` so definiert ist:

⟨*Listendefinition*⟩≡

```
liste([]).
liste([S|T]) :- liste(T).
```

Beachte, daß keine Einschränkung an S gemacht wurde.

Also: `[]` ist eine Liste, und wenn t eine Liste ist, so ist auch $[s|t]$ eine Liste: die Erweiterung der Liste t um den Term s.

Vereinfachte Schreibweisen:

$$\begin{array}{lll} [t_1, ..., t_n|t] & \text{statt} & [t_1|[t_2|\ldots[t_n|t]\ldots]] \\ [t_1, ..., t_n] & \text{statt} & [t_1, t_2, \ldots, t_n|[]] \end{array}$$

Beispiel `[s(0)|[]]` und `[0|[s(0)|[]]]` sind Listen, in der Kurzschreibweise `[s(0)]` und `[0,s(0)]`.

⟨*Gleichheitstest*⟩≡

```
?- [a|b] = '.'(a,b).
true.
?- '.'(a,b) = [a,b].
false.
```

```
?- '.'(a,b) = [a,[b]].
false.
?- '.'(a,[b]) = [a,b].
true.
```

Will man nur *homogene Listen*, bei denen alle Elemente gleichartig sind, z.B. nur Listen von natürlichen Zahlen, so kann man eine speziellere Definition verwenden:

⟨*Zahlenliste*⟩≡

```
natliste([]).
natliste([S|T]) :- nat(S), natliste(T).

nat(X) :- integer(X), X >= 0.
```

Mit Unifikation kann man sich Elemente aus einer Liste holen:

⟨*Zerlegung einer Liste*⟩≡

```
?- [_,_,X3,_|Rest] = [a,b,c,d,e,f,g].
X3 = c,
Rest = [e, f, g].
```

Umkehrung einer Liste:

`reverse([t1,...,tn],[tn,...,t1])`

Rekursive Implementierung:

Idee:

- Entferne das erste Element,
- kehre die (kürzere) Restliste um, und
- füge das erste Element hinten an die umgekehrte Restliste.

⟨*rekursive Listenumkehrung*⟩≡

```
reverse([],[]).
reverse([X|L],R) :-
        reverse(L,RL),
        append(RL,[X],R).

append([],L2,L2) :- liste(L2). % ggf. ohne liste(L2)
append([X|L1],L2,[X|L]) :-
        append(L1,L2,L).
```

Aufwand:

- `append(L1,L2,L)` macht $O(|L_1| + |L_2|)$ Schritte.
- Ist $f(|L|)$ die Anzahl der Schritte für `reverse(L,RL)`, so braucht man für `reverse([X|L],R)`

$$\begin{aligned} f(|L|+1) &= f(|L|) + O(|RL| + |[X]|) \\ &= f(|L|) + O(|L|+1) \\ &= \sum_{n=0}^{|L|+1} n = O((|L|+1)^2) \end{aligned}$$

  Schritte.

Der Algorithmus ist *quadratisch* in der Größe der Eingabe.

Iterative Implementierung mit Akkumulator

Idee:

- Staple die Terme der Eingabeliste der Reihe nach auf eine Liste von Zwischenergebnissen (Akkumulator),
- Beginne mit leerem Stapel,
- Gib den Stapel aus, wenn die Eingabeliste leer ist.

Programm:

⟨*iterative Listenumkehrung*⟩≡

```
reverse(L,R) :- rev(L,[],R).

rev([X|L],Acc,R) :-
        rev(L,[X|Acc],R).
rev([],Acc,Acc).
```

Aufwand:

- Für `rev([],Acc,Acc)` braucht man nur einen Schritt,

$$f(|[]|, |Acc|) = f(0, |Acc|) = 1.$$

- Ist $f(|L|, |[X|Acc]|)$ der Aufwand für `rev(L,[X|Acc],R)`, so ist der Aufwand für `rev([X|L],Acc,R)`

$$f(|L| + 1, |Acc|) = 2 + f(|L|, 1 + |Acc|) = 2 \cdot (|L| + 1) + 1$$

Also ist der Gesamtaufwand von `reverse(L,R)` hier

$$1 + f(|L|, 0) = 2 \cdot |L| + 2 = O(|L|).$$

Der Algorithmus ist *linear* in der Größe der Eingabe!

Komplexitätsmaß von Funktionen

Eine Funktion $f : \mathbb{N} \to \mathbb{N}$ *ist* (fast überall) *eine obere Schranke für die Zeitkomplexität von* $g : \mathbb{N} \to \mathbb{N}$, kurz $g \in O(f)$ oder "$g(n)$ ist $O(f(n))$", wenn

$$(*) \quad \exists n_0, c_0, c_1 \forall n \geq n_0 \, (Zeit(g, n) \leq c_1 \cdot f(n) + c_0)$$

wobei

$$Zeit(g, n) \quad := \quad \begin{cases} \text{Anzahl der Berechnungsschritte} \\ \text{zur Bestimmung von } g(n). \end{cases}$$

Die Bedingung $(*)$ sagt:

Für fast alle n ist $Zeit(g, n) \leq c_1 \cdot f(n) + c_0$.

Ein *Maschinenmodell* muß angeben, was ein Rechenschritt ist bzw. welche Grundfunktionen man in "einem" Schritt ausführen kann.

- $O(1)$: Funktionen, die mit konstantem Zeitaufwand berechnet werden können
- $O(n)$ (bzw. $O(n^2)$, $O(n^3)$, ...): Funktionen, die in linearem (bzw. quadratischem, kubischem,...) Zeitaufwand berechnet werden können
- $O(2^n)$: Funktionen, die mit exponentiellem Zeitaufwand berechnet werden können.

Hat g andere Argument- und Wertbereiche als $\mathbb{N}$, z.B. $g : Term \to Term$, so braucht man ein Maß $|\cdot|: Term \to \mathbb{N}$ für die Ein- und Ausgabewerte und nimmt statt (*)

$$(**) \quad \exists n_0, c_0, c_1 \forall t(|t| \geq n_0 \Rightarrow (Zeit(g,t) \leq c_1 \cdot f(|t|) + c_0)).$$

Sortieren mit Quicksort

Sortiere eine Liste L von Zahlen nach ihrer Größe zu L'. Idee:

1. Wähle ein Element X aus L;
2. Zerlege L in eine Liste *Big* aller Elemente $Y \geq X$ und eine Liste *Small* aller Elemente $Y < X$;
3. Sortiere rekursiv die Listen *Big* und *Small* zu *Big*', *Small*';
4. Bilde L' durch $append(Big', [X|Small'], L')$.

⟨*Quicksort rekursiv*⟩≡

```
/* quicksort(+Zahlliste,-Sortiert) */
quicksort([],[]).
quicksort([X|Xs],Ys) :-
        partition(X,Xs,Big,Small),
        quicksort(Big,Bs),
        quicksort(Small,Ss),
        append(Bs,[X|Ss],Ys).
```

⟨*partition*⟩≡

```
/* partition(+X,+List,-Big,-Small) */

partition(_,[],[],[]).
partition(X,[Z|Zs],ZsBig,[Z|ZsSmall]) :-
        Z < X, partition(X,Zs,ZsBig,ZsSmall).
partition(X,[Z|Zs],[Z|ZsBig],ZsSmall) :-
        Z >= X, partition(X,Zs,ZsBig,ZsSmall).
```

Der Vergleich von Zahlen wird durch vordefinierte Prolog-Prädikate < und >= vorgenommen.

Quicksort mit Akkumulator

Vermeide die Aufrufe von `append`: akkumuliere alle schon gesehenen Elemente der Eingabe (sortiert, die kleinen vorn!):

⟨*Quicksort mit Akkumulator*⟩≡

```
/* quicksort(Liste,Sortiert) */
quicksort(Xs,Ys) :-
        qusort(Xs,[],Ys).

/* qusort(Liste,Acc,Sortiert),
   Elemente in Acc sind > die in Liste */
qusort([],Acc,Acc).
qusort([X|Xs],Acc,Ys) :-
        partition(X,Xs,Big,Small),
        qusort(Big,Acc,Zs),
        qusort(Small,[X|Zs],Ys).
```

Korrektheit: Man zeigt durch Induktion über n: falls

$$qusort([t_1, \ldots, t_n], [s_1, \ldots, s_m], [r_1, \ldots, r_k]),$$

so ist $k = n + m$ und $[r_1, \ldots, r_k] = [t'_1, \ldots, t'_n, s_1, \ldots, s_m]$, wobei $[t'_1, \ldots, t'_n]$ die sortierte Liste der t_i ist, mit kleinstem Element links.

⟨*Trace von Quicksort mit Akkumulator*⟩≡

```
?- trace.
?- quicksort([2,3,0,1],Ys).
Call:( 8) quicksort([2,3,0,1],_G242) ?
Call:( 9) qusort([2,3,0,1],[],_G242) ?
Call:(10) partition(2,[3,0,1],_L144,_L145) ? s
Exit:(10) partition(2,[3,0,1],[3],[0,1]) ?
Call:(10) qusort([3],[],_L146) ? s
Exit:(10) qusort([3],[],[3]) ?
Call:(10) qusort([0,1],[2,3],_G242) ?
Call:(11) partition(0,[1],_L252,_L253) ? s
Exit:(11) partition(0,[1],[1],[]) ?
Call:(11) qusort([1],[2,3],_L254) ?
Call:(12) partition(1,[],_L298,_L299) ? s
Exit:(12) partition(1,[],[],[]) ?
```

⟨*Fortsetzung des Traces*⟩≡

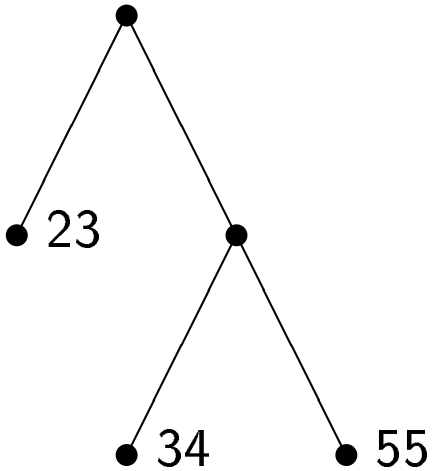
```
Call:(12) qusort([],[2,3],_L300) ?
Exit:(12) qusort([],[2,3],[2,3]) ?
Call:(12) qusort([],[1,2,3],_L254) ? s
Exit:(12) qusort([],[1,2,3],[1,2,3]) ?
Exit:(11) qusort([1],[2,3],[1,2,3]) ?
Call:(11) qusort([],[0,1,2,3],_G242) ?
Exit:(11) qusort([],[0,1,2,3],[0,1,2,3]) ?
Exit:(10) qusort([0,1],[2,3],[0,1,2,3]) ?
Exit:( 9) qusort([2,3,0,1],[],[0,1,2,3]) ?
Exit:( 8) quicksort([2,3,0,1],[0,1,2,3]) ?

Ys = [0,1,2,3]
```

Was muß man ändern, damit die großen Elemente links stehen?

Datentypen in Prolog

Wie das Beispiel der natürlichen Zahlen zeigte, muß der Programmierer seine Daten durch *Terme* von Prolog darstellen.

Beispiel Binäre Bäume mit Zahlen an den Blättern, etwa



können wir als Terme auffassen, indem wir das Funktionszeichen '.'/2 für Verzweigungsknoten benutzen:

$$'.'(23,'.'(34,55))$$

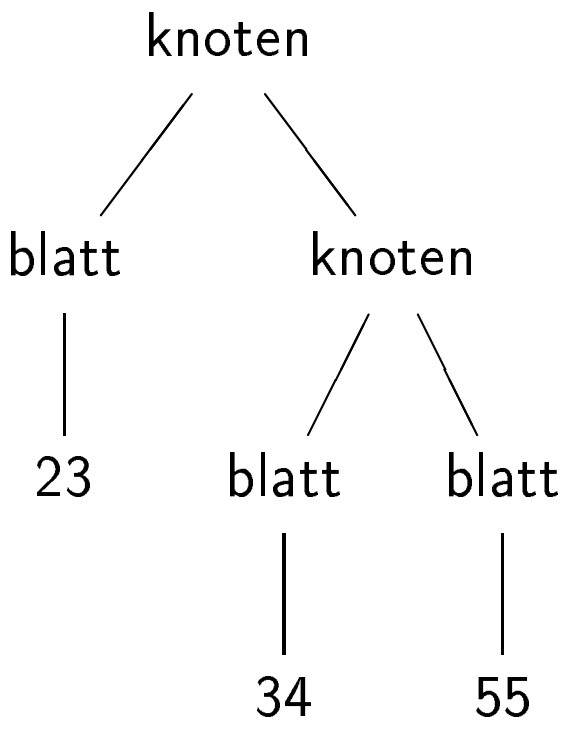
In Prolog haben wir dafür eine andere Notation:

⟨*Prolognotation*⟩≡

```
33 ?- X = '.'(23,'.'(34,55)).

X = [23, 34|55]
```

Diese Notation ist aber nicht besonders lesbar und führt leicht zu Verwechslungen mit dem Spezialfall der Listen.

Besser ist es, wenn man eigene Funktionszeichen benutzt und den Baum etwa so schreibt:



Wir unterscheiden *Verzweigungsknoten* von *Blättern* des Baums. Dazu verwenden wir die Funktionszeichen `knoten/2` und `blatt/1`.

Die Prolog-Terme, die wir zur Darstellung von Bäumen verwenden, legen wir durch ein Prolog-Prädikat `baum/1` fest:

⟨*baum.pl*⟩≡

```
baum(blatt(N)) :- number(N).
baum(knoten(X,Y)) :- baum(X), baum(Y).
```

Der Beispielbaum ist also durch folgenden Term dargestellt:

⟨*Beispielbaum*⟩≡

```
knoten(blatt(23),knoten(blatt(34),blatt(55)))
```

Nun braucht man Prädikate, die die Daten dieser selbstdefinierten Datentypen erkennen und verarbeiten.

Prädikat `blaetter(+Baum,-Atomliste)`

Wir sammeln die Blattmarken eines Baums in einer Liste:
Rekursiv geht das einfach so:

⟨*Rekursives Sammeln*⟩≡

```
blaetter(knoten(Links,Rechts), Marken) :-
        blaetter(Links, MarkenL),
        blaetter(Rechts, MarkenR),
        append(MarkenL,MarkenR, Marken).
blaetter(blatt(Marke), [Marke]).
```

Effizienter ist es, wenn wir die schon gesehenen Marken auf einer anfangs leeren Hilfsliste `Acc` sammeln:

⟨*Sammeln mit Akkumulator*⟩≡

```
blaetter(Baum, Marken) :-
        blaetter(Baum,[], Marken).

blaetter(knoten(Links,Rechts),Acc, Marken) :-
        blaetter(Rechts,Acc, RAcc),
        blaetter(Links,RAcc, Marken).
blaetter(blatt(Marke), Acc, [Marke|Acc]).
```

Auf den Stapel `Acc` kommen *zuerst* die Marken des *rechten* Teilbaums, damit die des linken vorne auf der Markenliste stehen!

⟨*Beispielaufruf*⟩≡

```
?- blaetter(knoten(blatt(23),
                   knoten(blatt(34),blatt(55))),
            Marken).
Marken = [23,34,55]
```

Beschneiden des Suchbaums durch !

Man kann die Suche nach alternativen Beweisen durch das 0-stellige, stets beweisbare Prädikat **!** ('cut') abschneiden:

⟨*Beispiel*⟩≡

```
praedikat(Xs) :-
        p1(Xs,Ys),..,pn(Xs,Ys),
        !,
        q1(Xs,Ys,Zs),...,qk(Xs,Ys,Zs).
praedikat(Xs) :-
        r(Xs,Ys).
```

Bedeutung von !

1. Um `praedikat(Us)` zu beweisen, wird durch das Beweisen von

   `p1(Xs',Ys'),...,pn(Xs',Ys')`

   eine Belegung S mit $\texttt{Xs}'S = \texttt{Us}S$ aufgebaut;
2. diese Belegung wird durch `!` als *die einzige (bzw. letzte) erlaubte Belegung* der `Xs',Ys'` erklärt, die für den Beweis von

   `praedikat(Us)`

   in Frage kommt.
3. Alternativen werden für die `q1,...,qk` gesucht (ggf. mit verschiedenen Belegungen der `Zs`), aber nicht für die `p1,...,pn`;
4. Es wird keine weitere Alternative für `praedikat(Us)`, z.B. über `r(Xs,Ys)`, probiert.

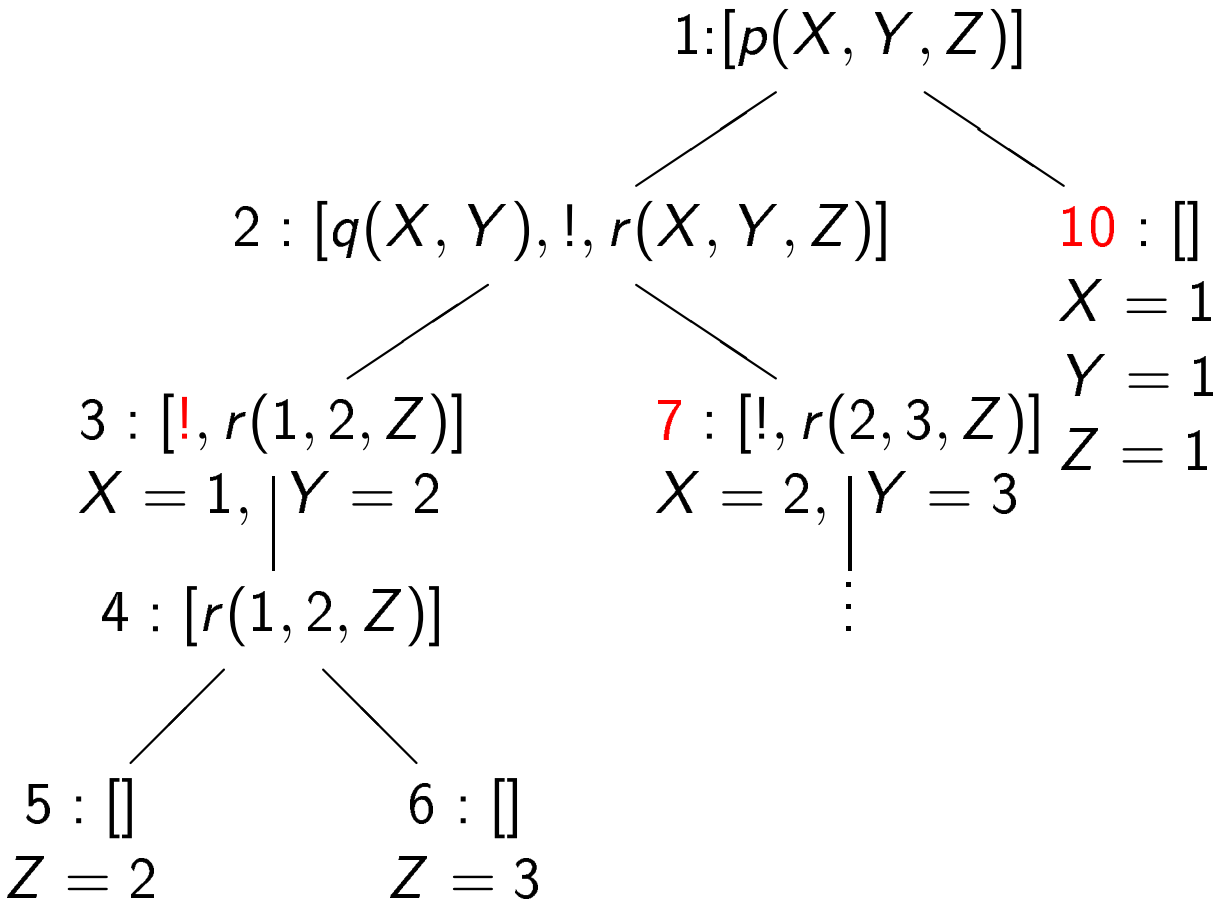
Beweissuche und !

Den Effekt eines ! zeigt folgendes Beispiel.

⟨*Beispiel*⟩+≡

```
p(X,Y,Z) :- q(X,Y), !, r(X,Y,Z).
p(1,1,1).

q(1,2).
q(2,3).
r(1,2,2).
r(1,2,3).
r(2,3,3).
```

Wäre `!` wie `true`, so wäre der Suchbaum zu `?- p(X,Y,Z)` so:



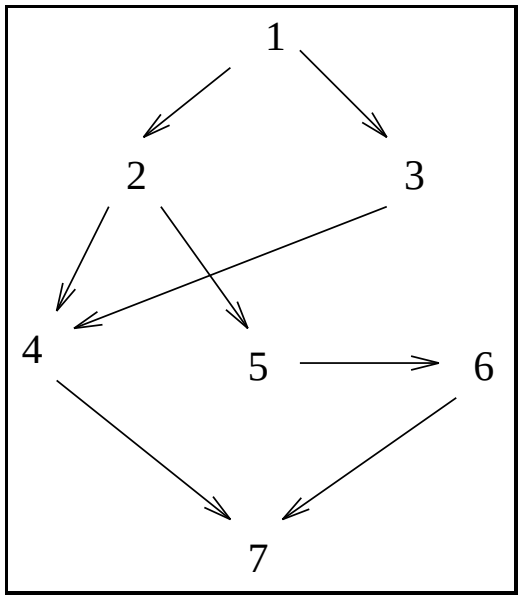
Durch ! beim Übergang zu Ziel 4 werden die Alternativen 7 und 10 abgeschnitten. Nur (X,Y,Z) = (1,2,2) und (1,2,3) sind Lösungen!

Beispiel für das Abschneiden der Suche

Sei $\mathcal{G} = (\{1, \ldots, m\}, \mathit{kante})$ ein kreisfreier Graph, etwa wieder

$\langle \mathit{Kanten} \rangle \equiv$

```
kante(1,2).
kante(1,3).
kante(2,4).
kante(2,5).
kante(3,4).
kante(4,7).
kante(5,6).
kante(6,7).
```



mit der folgenden Definitionen von Wegen:

```
weg(X,Y) :- kante(X,Y).
weg(X,Y) :- kante(X,Z), weg(Z,Y).
```

a) Um festzustellen, ob Y von X erreichbar ist, genügt es, *einen* Weg von X nach Y zu finden:

⟨*Weg von X zu Y*⟩≡

```
erreichbar(X,Y) :-
        weg(X,Y), !.
```

b) Falls ein Weg von X nach 3 führt, suche nur die von X über 3 erreichbaren Y, im anderen Fall alle von X erreichbaren Y:

⟨*Weg von X zu Y über 3*⟩≡

```
erreichbar3(X,Y) :-
        weg(X,3), !, weg(3,Y).
erreichbar3(X,Y) :-
        weg(X,Y).
```

Antworten:

a) Man kann alle erreichbaren Punkte erreichen, z.B.:

$\langle$*Weg von X zu Y*$\rangle$+$\equiv$

```
?- erreichbar(1,7).
Yes
?- erreichbar(1,5).
Yes
```

Aber `erreichbar/1` taugt nicht zum *Suchen* erreichbarer Punkte: es stoppt beim ersten erreichten Punkt:

⟨*Weg von X zu Y*⟩+≡

```
?- erreichbar(1,Y).
Y = 2 ;
No
?- erreichbar(X,7).
X = 4 ;
No
```

b) Da von 1 ein Weg nach 3 führt, werden nur „dessen“ Fortsetzungen nach 4 und 7 gefunden, die Wege von 1 über 2 nach 4,5,6,7 aber nicht:

⟨*Weg von X zu Y über 3*⟩+≡

```
?- erreichbar3(1,Y).
Y = 4 ;
Y = 7 ;
No
```

Da von 2 kein Weg nach 3 führt, findet man alle Wege:

⟨*Weg von X zu Y über 3*⟩+≡

```
?- erreichbar3(2,Y).
Y = 4 ;
Y = 5 ;
Y = 7 ;
Y = 6 ;
Y = 7 ; No
```

Fallunterscheidung mit !

Wegen ! braucht man in `fall2(X)` nicht einzubauen, daß `not(fall1(X))` gelten muß. Analog für die späteren Fälle.

⟨*Fallunterscheidung*⟩≡

```
tue_es(X,Y) :-
        fall1(X),!,antwort1(X,Y).
tue_es(X,Y) :-
        fall2(X),!,antwort2(X,Y).
tue_es(X,Y) :-
        fall3(X),!,antwort3(X,Y).
tue_es(X,Y) :-
        andernfalls(X,Y).
```

Beachte auch, daß keine alternativen Beweise für die einzelnen Fälle gesucht werden!

⟨*trenne/3 ohne !*⟩≡

```
trenne([],[],[]).
trenne([X|Xs],[X|Zs],Ns) :-
    number(X),                  % Fall 1
    trenne(Xs,Zs,Ns).
trenne([X|Xs],Zs,[X|Ns]) :-
    not(number(X)),             % Fall 2: sonst
    trenne(Xs,Zs,Ns).           % prueft number(X) nochmal
```

⟨*trenne/3 mit !*⟩≡

```
trenne([],[],[]).
trenne([X|Xs],[X|Zs],Ns) :-
    number(X),                  % Fall 1
    !, trenne(Xs,Zs,Ns).
trenne([X|Xs],Zs,[X|Ns]) :-
    trenne(Xs,Zs,Ns).           % Sonst-Fall
```

Beispiel

Wie oft kommt ein Term in einer Liste vor? (Eingabeargumente von `oftIn(+Term,+Liste,-Zahl)` *sollten keine Variable enthalten.)*

⟨*Termzählen*⟩≡

```
oftIn(T,[X|L],N) :- X=T, !, oftIn(T,L,M), N is M+1.
oftIn(T,[X|L],N) :-
        oftIn(T,L,N).
oftIn(T,[],0).
```

Ohne `!` *bekäme man falsche Antworten durch den 2. Fall.*

Abschneiden von Alternativen: Deterministische Programme

Vereinige zwei (sortierte) Listen von Zahlen so, daß die Ergebnisliste sortiert ist:

⟨*Vereinigung*⟩≡

```
merge1([X|Xs],[Y|Ys],[X|Zs]) :-
        X < Y, merge1(Xs,[Y|Ys],Zs).
merge1([X|Xs],[Y|Ys],[X,Y|Zs]) :-
        X = Y, merge1(Xs,Ys,Zs).
merge1([X|Xs],[Y|Ys],[Y|Zs]) :-
        X > Y, merge1([X|Xs],Ys,Zs).
merge1([],Ys,Ys).
merge1(Xs,[],Xs).
```

Das Programm berücksichtigt Alternativen, die nichts zur Lösung beitragen, weil sie scheitern oder eine Lösung wiederholen. Das kostet Zeit und Platz bei der Suche!

Abschneiden unnötiger Alternativen macht das Programm deterministisch (Suchbaum ohne Verzweigungspunkte):

⟨*deterministische Vereinigung*⟩≡

```
merge2([X|Xs],[Y|Ys],[X|Zs]) :-
        X < Y, !, merge2(Xs,[Y|Ys],Zs).
merge2([X|Xs],[Y|Ys],[X,Y|Zs]) :-
        X = Y, !, merge2(Xs,Ys,Zs).
merge2([X|Xs],[Y|Ys],[Y|Zs]) :-
        X > Y, !, merge2([X|Xs],Ys,Zs).
merge2([],Ys,Ys) :- !.
merge2(Xs,[],Xs).
```

Bem.: Jetzt hat auch `merge([],[],Zs)` nur eine Lösung.

Trace zu merge1

⟨*Trace*⟩≡

```
17 ?- trace.
Yes
17 ?- merge1([0,2],[1,3],L).
   Call: (6) merge1([0, 2], [1, 3], _G398) ?
   Call: (7) 0<1 ?
   Exit: (7) 0<1 ?
   Call: (7) merge1([2], [1, 3], _G467) ?
   Call: (8) 2<1 ?
   Fail: (8) 2<1 ?
   Redo: (7) merge1([2], [1, 3], _G467) ?
   Call: (8) 2=1 ?
   Fail: (8) 2=1 ?
   Redo: (7) merge1([2], [1, 3], _G467) ?
```

⟨*Rest des Traces*⟩≡

```
Call: (8) 2>1 ?
Exit: (8) 2>1 ?
Call: (8) merge1([2], [3], _G473) ?
Call: (9) 2<3 ?
Exit: (9) 2<3 ?
Call: (9) merge1([], [3], _G479) ?
Exit: (9) merge1([], [3], [3]) ?
Exit: (8) merge1([2], [3], [2, 3]) ?
Exit: (7) merge1([2], [1, 3], [1, 2, 3]) ?
Exit: (6) merge1([0, 2], [1, 3], [0, 1, 2, 3]) ?

  L = [0, 1, 2, 3] ;
```

Gibt es weitere Lösungen?

Trace zu merge1 (Forts.)

Es gibt zwar keine zweite Lösung, aber die Suche danach kostet Arbeit, bevor man das feststellt:

⟨*Suche nach 2.Lösung*⟩≡

```
Redo: (9) merge1([], [3], _G479) ?
Fail: (9) merge1([], [3], _G479) ?
Redo: (8) merge1([2], [3], _G473) ?
Call: (9) 2=3 ?
Fail: (9) 2=3 ?
Redo: (8) merge1([2], [3], _G473) ?
Call: (9) 2>3 ?
Fail: (9) 2>3 ?
Redo: (8) merge1([2], [3], _G473) ?
Fail: (8) merge1([2], [3], _G473) ?
```

⟨*weitere Redos*⟩≡

```
    Redo: (7) merge1([2], [1, 3], _G467) ?
    Fail: (7) merge1([2], [1, 3], _G467) ?
    Redo: (6) merge1([0, 2], [1, 3], _G398) ?
    Call: (7) 0=1 ?
    Fail: (7) 0=1 ?
    Redo: (6) merge1([0, 2], [1, 3], _G398) ?
    Call: (7) 0>1 ?
    Fail: (7) 0>1 ?
    Redo: (6) merge1([0, 2], [1, 3], _G398) ?
    Fail: (6) merge1([0, 2], [1, 3], _G398) ?

No
18 ?-
```

Trace zu merge2

Man findet dieselbe Lösung, vermeidet aber die Arbeit der Suche nach weiteren (nicht existierenden) Lösungen:

⟨*Trace*⟩+≡

```
18 ?- trace.
Yes
18 ?- merge2([0,2],[1,3],L).
   Call: (6) merge2([0, 2], [1, 3], _G398) ?
   Call: (7) 0<1 ?
   Exit: (7) 0<1 ?
   Call: (7) merge2([2], [1, 3], _G467) ?
   Call: (8) 2<1 ?
   Fail: (8) 2<1 ?
   Redo: (7) merge2([2], [1, 3], _G467) ?
   Call: (8) 2=1 ?
```

⟨*Trace*⟩+≡

```
    Fail: (8) 2=1 ?
    Redo: (7) merge2([2], [1, 3], _G467) ?
    Call: (8) 2>1 ?
    Exit: (8) 2>1 ?
    Call: (8) merge2([2], [3], _G473) ?
    Call: (9) 2<3 ?
    Exit: (9) 2<3 ?
    Call: (9) merge2([], [3], _G479) ?
    Exit: (9) merge2([], [3], [3]) ?
    Exit: (8) merge2([2], [3], [2, 3]) ?
    Exit: (7) merge2([2], [1, 3], [1, 2, 3]) ?
    Exit: (6) merge2([0, 2], [1, 3], [0, 1, 2, 3]) ?
L = [0, 1, 2, 3] ;
No
19 ?-
```

Fallunterscheidung

Für die Fallunterscheidung gibt es einen eigenen Ausdruck,

⟨*Fallunterscheidung*⟩+≡

```
(If -> Then ; Else).
```

Er hat die Bedeutung:

- Versuche, die Klausel `If` zu beweisen.
- Falls das gelingt, beweise die Klausel `Then`,
- sonst beweise die Klausel `Else`.

Beim Rücksetzen sucht man *keinen zweiten Beweis für* `If`.

Man kann sich das selbst wie folgt mit `!` definieren:

⟨*Definition der Fallunterscheidung*⟩≡

```
if(If,Then,Else) :-
        If, !, Then.
if(If,Then,Else) :-
        Else.
```

Beispiel

Fallunterscheidung bei `merge` *mit* `->` *formuliert:*

⟨*merge3*⟩≡

```
merge3([], A, A) :- !.
merge3(A, [], A) :- !.

merge3([A|B], [C|D], [E|F]) :-
        (   A@=<C
        ->  E=A,
            merge3(B, [C|D], F)
        ;   E=C,
            merge3([A|B], D, F)
        ).
```

Das ist das `merge` *in SWI-Prolog.*

⟨*Trace zu merge3*⟩≡

```
27 ?- trace.
Yes
27 ?- merge3([0,2],[1,3],L).
   Call: (6) merge3([0, 2], [1, 3], _G398) ?
   Call: (7) 0@=<1 ?
   Exit: (7) 0@=<1 ?
   Call: (7) _G466=0 ?
   Exit: (7) 0=0 ?
   Call: (7) merge3([2], [1, 3], _G467) ?
   Call: (8) 2@=<1 ?
   Fail: (8) 2@=<1 ?
   Call: (8) _G472=1 ?
   Exit: (8) 1=1 ?
   Call: (8) merge3([2], [3], _G473) ?
```

⟨*Rest des Traces*⟩+≡

```
    Call: (9) 2@=<3 ?
    Exit: (9) 2@=<3 ?
    Call: (9) _G478=2 ?
    Exit: (9) 2=2 ?
    Call: (9) merge3([], [3], _G479) ?
    Exit: (9) merge3([], [3], [3]) ?
    Exit: (8) merge3([2], [3], [2, 3]) ?
    Exit: (7) merge3([2], [1, 3], [1, 2, 3]) ?
    Exit: (6) merge3([0, 2], [1, 3], [0, 1, 2, 3]) ?
L = [0, 1, 2, 3] ;
    Fail: (8) merge3([2], [3], _G473) ?
    Fail: (7) merge3([2], [1, 3], _G467) ?
    Fail: (6) merge3([0, 2], [1, 3], _G398) ?
No
28 ?-
```

Schleifen

Um eine Schleife in Prolog zu simulieren, benutzt man folgenden Umweg über die Beweissuche:

⟨*Schleife*⟩≡

```
p(X) :-
        q(X,Y),
        fail.
p(X).
```

- Da `fail` nicht bewiesen werden kann, muß durch Rücksetzen *jede* Lösung `Y` von `q(X,Y)` gesucht werden.
- Schließlich kommt man durch Rücksetzen zur zweiten Klausel von `p(X)`, wodurch die Schleife beendet wird.

⟨*andere Schleifenschreibweise*⟩≡

```
p(X) :- ( q(X,Y), fail ; true ).
```

Beispiel Anzeigen eines Graphen `kante/2`:

⟨*Graphenanzeige*⟩≡

```
zeigeGraph :-
        kante(X,Y),    /* mehrere Loesungen! */
        write('Kante von Punkt '),
        write(X),
        write(' nach Punkt '),
        write(Y),
        write('.'),
        nl,            /* newline: neue Zeile */
        fail.
zeigeGraph.
```

Termeigenschaften

Prolog hat eingebaute Konstante, mit denen man Eigenschaften eines Prolog-Terms prüfen kann:

1. `var(X)` gilt, wenn die Belegung von `X` eine Variable ist,
2. `nonvar(X)` gilt, wenn die Belegung von `X` keine Variable ist,
3. `ground(X)` gilt, wenn die Belegung von `X` keine freien Variablen enthält,
4. `compound(X)` gilt, wenn die Belegung von `X` ein zusammengesetzter Term ist,
5. `atom(X)` gilt, wenn die Belegung von `X` ein Prolog-Atom ist,
6. `number(X)` gilt, wenn die Belegung von `X` eine (reelle) Zahl ist,
7. `atomic(X)` gilt, wenn die Belegung von `X` ein Atom oder eine (reelle) Zahl ist.

Beispiel zum Prüfen, ob eine Liste nur Grundterme enthält:

```
groundlist([]).
groundlist([X|L]) :-
    ground(X), groundlist(L).
```

Termzerlegung

Man kann mit

```
functor(+Term,-Funktionssymbol,-Stelligkeit)
arg(+Stelle,+Term,-Teilterm)
```

einen Prolog-Term in Funktionssymbol und Argumente zerlegen:

```
?- functor(f(a,b,c,d,e),F,N).
F = f
N = 5
?- arg(3,f(a,b,c,d,e),A).
A = c
```

Der Operator 'univ': =..

Termzerlegung und Termzusammensetzung kann man auch mit =.. durchführen:

Zerlegung

Auf die Anfrage

```
?- f(t1,...,tn) =.. X.
```

liefert Prolog die Antwort

```
X = [f,t1,...,tn]
```

Zusammensetzung

Auf die Anfrage

```
?- X =.. [f,t1,...,tn].
```

antwortet Prolog mit:

```
X = f(t1,...,tn),
```

falls `f` ein Atom ist, sonst scheitert die Anfrage.

Wegen `[s|t] = '.'(s,t)` und `[a,b,c] = [a|[b,c]]` erhält man

```
?- [a,b,c] =.. [Funktor|Argumente].
Funktor = '.',
Argumente = [a, [b, c]].
```

Man muß „univ“ `=..` von der „Unifikation“ `=` unterscheiden:

```
?- [a,b,c] = [F|Args].
F = a
Args = [b,c]
```

Man kann auch Terme mit variablen Argumenten zusammensetzen:

```
?- Term =.. [f,a,X,b].
Term = f(a,X,b)
```

Beispiel: Wert eines Terms im Termmodell

Substitution oder der Wert von `Term` bei einer Belegung

```
Env = [(X1,V1),...(Xn,Vn)]
```

```
value(X,Env,Val) :-
   var(X),
   !, lookup(X,Env,Val). % Fall Variable zuerst!
value(Term,Env,Val) :-
   atomic(Term),
   !, Val = Term.
value(Term,Env,Val) :-
   Term =.. [F|Args],    % Zerlegen
   !, values(Args,Env,Vals),
   Val =.. [F|Vals].     % Zusammensetzen
```

```
lookup(X,[(Y,V)|Env],Val) :-
   !, ( X == Y -> Val = V ; lookup(X,Env,Val) ).
lookup(X,[],X).

values([T|Ts],Env,[V|Vs]) :-
   value(T,Env,V), values(Ts,Env,Vs).
values([],Env,[]).
```

Beispiel

```
?- value(f(X,Y),[(Z,3),(X,g(X,Y))],Val).
Val = f(g(X, Y), Y).
```

Bem. Das entspricht der früher verwendeten Notation

```
f(X,Y){3/Z, g(X,Y)/X} = f(g(X,Y),Y)
```

STRÖME zur Ein- und Ausgabe

Der Prolog-Prozeß kommuniziert mit der Außenwelt über *Ströme*:

- Daten werden von *Eingabeströmen* gelesen,
- Daten werden auf *Ausgabestöme* geschrieben.

Erzeugen eines Stroms

Man erzeugt und *öffnet* einen Strom zum Lesen oder Schreiben mit

```
open(+SrcDest, +Mode, -Stream, +Options)
```

wobei

- `SrcDest`: ein Atom (Dateiname) oder Term der Form `pipe(Command)`, z.B. `pipe(lpr)` = zum Drucker.
- `Mode`: `read` zum Lesen am Beginn, `write` zum Löschen und dann Schreiben am Beginn, `append` zum Schreiben am Ende, `update` zum Überschreiben am Anfang.

- `Stream`: ein Identifikator ('$stream'(<Nummer>) des Stroms
- `Options`: eine Liste von Optionen, z.B. `type(binary)`, `type(text)`, `eof_action(Action)`, `buffer(Bmode)`.

  `buffer(fullf)`,`buffer(line)`,`buffer(false)` gibt an, ob bzw. wieviel Text im Arbeitsspeicher gepuffert wird, bevor das Schreiben auf dem Strom erfolgt.

  Puffer werden mit `flush/0`,`flush_output/1` geleert.

Schließen eines Stroms

Man *schließt* den Strom `Stream` (und leert den Puffer) durch

```
close(Stream).
```

Implizite Stöme

Was der jeweils *aktuelle* Strom ist, der implizit von `read/1`, `write/1` benutzt wird, kann mit

```
?- current_input(X), current_output(Y).
```

abgefragt (und ihre Namen in `X,Y` gespeichert) werden.

Zu den Prädikaten `write/1`, `read/1` gibt es Varianten `write(+Strom,+Term)`, `read(+Strom,-Term)` mit explizitem Strom.

Statt mit `open/4` einen Strom zu erzeugen, zu benennen und einer Datei (einem Prozeß) zuzuordnen, kann man einfach den aktuellen Lese- oder Schreibstrom wechseln:

Lesen von einer Datei

```
see(Datei),     % akt.Eingabestrom wechseln
...
read(Term),     % liest jetzt von Datei
...
seen.           % akt.Eingabestrom wie vorher
```

Schreiben auf eine Datei

```
tell(Datei),    % akt.Ausgabestrom wechseln
...             % append(Datei): am Dateiende schreiben
write(Term),    % schreibt jetzt auf Datei
...
told.           % akt.Ausgabestrom wie vorher
```

Die Datei mit Namen `Datei` (Prolog-Atom) sollte schon existieren.

Dialog mit dem Benutzer

erfolgt über die Ströme `user` bzw. `user_input`, `user_output`.

Mitteilung an den Benutzer

Aus einem Programm kann man etwas an den Bildschirm schreiben:

```
write(user,+Term)
```

Antwort des Benutzers aufnehmen

Ein Programm kann mit `read(user,-Term)` den Benutzer am Bildschirm um eine Information fragen:

```
?- read(user,Term).
|: f(a,g(b)).
Term = f(a, g(b)).
```

Die Eingabeaufforderung lautet `|:` statt `?-` und die Eingabe muß mit `'.'` abgeschlossen werden.

⟨*alter.pl*⟩≡

```
altersAbfrage :-
   write(user,'Wie alt sind Sie?'),
   write(user,'(Antworten Sie mit <Anzahl>.) '),
   read(user,N),
   ((7 =< N, N < 100)
   -> write(user,'Gut, das glauben wir.')
   ; write(user,'Erstaunlich. Wir gratulieren!')),
      write(user,'\nUnd wie heissen Sie? '),
      write(user,'(Antworten Sie mit \'<Name>\'.) '),
      read(user,Name),
      append(altersDatei)
      write('Name: '),writeq(Name),nl,
      write('Alter: '), write(N),nl,nl,
      told,
      write(user,'Danke, das war alles.'
   ).
```

Anwendung: Tokenizer

1. Den nächsten Satz lesen und als Zeichenfolge ausgeben:
   `read_sentence(+Stream,-Sentence,+Accumulator)`
   - Lies Zeichenweise von einer Datei (oder der Konsole),
   - Sammle die Zeichen und höre beim nächsten Satzende auf.

2. Die Zeichenfolge in eine Wortfolge umwandeln:
   `stringToAtoms(+Sentence,-Atomlist)`
   - Zerlege die Zeichenfolge in einzelne Wortvorkommen (Token),
   - Wandle die Wörter in Prolog-Atome um,
   - Gib die entstandene Atomfolge aus.

Einlesen von Sätzen aus einer Datei

⟨*Aus einer Datei lesen*⟩≡

```
read_sentence(File,Atomlist) :-
        % Fall 1: File ist offen
        current_stream(File,read,Stream),!,
        read_sentence(Stream,Sentence,[]),
        stringToAtoms(Sentence,Atomlist).
read_sentence(File,Atomlist) :-
        % Fall 2: sonst
        open(File,read,Stream),
        read_sentence(Stream,Sentence,[]),
        stringToAtoms(Sentence,Atomlist).
```

Alternativ: Einlesen von der Konsole

Es ist nützlich, dem Benutzer zu sagen, wie die Eingabe zu beenden ist, und am Schluß ein Echo der erkannten Atomliste zu geben.

⟨*Von der Konsole lesen*⟩≡

```
eingabe :-
        write('Beende die Eingabe mit <.|?><Return>'),
        nl,read_sentence(user,Sentence,[]),
        stringToAtoms(Sentence,Atomlist),
        nl,writeq(Atomlist).
```

Satzende-Erkenner

Zeichenstrom $\mapsto$ Zeichen bis zum Satzende

Wir lesen von `Stream` und sammeln die gesehenen Zeichen in `Chars` `read_sentence(+Stream,-Sentence,+Chars)`, sammeln die gesehenen Zeichensein Hilfsprädikat `read_sentence/4`

⟨*read_sentence/3*⟩≡

```
read_sentence(Stream,Sentence,Seen) :-
        get0(Stream,Char),!,
        read_sentence(Stream,Sentence,Char,Seen).

% read_sentence(+Strom,-Sent.,+LastChar,+CharsRev)
read_sentence(Stream,Sentence,Char,Seen) :-
        dateiende(Char),
        !,reverse(Seen,Sentence),
        name(Atom,Sentence),nl,write(Atom), % Echo
        nl,close(Stream),
        nl,write('End of File reached; Stream closed.').
```

Alle ". ", die *keine* Satzenden darstellen, sind Sonderfälle und *vor* dem Normalfall zu behandeln:

⟨*Sonderfall*⟩≡

```
read_sentence(Stream,Sent,Char,[Pkt,Zif|Seen]) :-
        ordinalzahlende([Zif,Pkt,Char]),!,
        read_sentence(Stream,Sent,[Char,Pkt,Zif|Seen]).
read_sentence(Stream,Sent,Char,[R|Seen]) :-
        abkuerzungsendeRev([Char,R|Seen]),
        !, read_sentence(Stream,Sent,[Char,R|Seen]).
```

Sonst, im Normalfall, gilt ". " als Satzende, und man gibt die Zeichen des Satzes zurück:

⟨*Normalfall*⟩≡

```
read_sentence(Stream,Sentence,Char,[C|Seen]) :-
        satzende([C,Char]), % s.u.
        !, % Satzende gefunden, Satz ausgeben:
        reverse([C|Seen],Sentence).
```

Sonst, wenn die letzten Zeichen kein Satzende bilden, kommt das zuletzt gelesene Zeichen auf den Akkumulator und es wird mit `read_sentence/3` weitergelesen:

⟨*Weiterlesen*⟩≡

```
read_sentence(Stream,Sentence,Char,Seen) :-
        read_sentence(Stream,Sentence,[Char|Seen]).
```

Tokenizer:

Zeichenreihe $\mapsto$ Atomfolge

Vorgehensweise:

1. durchlaufe die Zeichenfolge von links nach rechts,
2. unterscheide dabei zwei Fälle:
   - 2.1 das nächste Zeichen beginnt evtl. ein Wort
   - 2.2 das nächste Zeichen setzt ein Wort fort
3. stapele die gelesenen Zeichen eines Wortanfangs
4. stapele die schon erkannten Wörter (als Atome),
5. am Ende gib den umgekehrten Stapel der Atome aus.

⟨*stringToAtoms(+String,-Atomlist)*⟩≡

```
stringToAtoms(String,Atomlist) :-
    tokenize(String,[],[],Atomlist).
```

```
tokenize(+String,+CharsRev,+TokensRev,-Tokens)
```

(a) Suche nach dem nächsten Wortanfang

⟨*Wortanfangssuche*⟩≡

```
tokenize([T|String],[],Rs,Ts) :-
    (leerZeichen(T); zeilenende(T)),
    % fuehrende Leerzeichen,Zeilenumbruch weglassen
    !,tokenize(String,[],Rs,Ts).
tokenize([T|String],[],Rs,Ts) :-
    sonderzeichen(Atom,T),  % eigenes Token
    !,tokenize(String,[],[Atom|Rs],Ts).
tokenize([C|String],[],Rs,Ts) :- % Wortanfang!
    !,tokenize(String,[C],Rs,Ts).% sammle Zeichen
```

(b) Weiterlesen in einem angefangenen Wort

Sonderfall: Worttrennung am Zeilenende aufheben

⟨*Leerzeichen am Zeilenanfang beseitigen*⟩≡

```
tokenize([N,L|String],[T|Cs],Rs,Ts) :-
    worttrennung([T,N]),leerZeichen(L),
    % Leerzeichen am Zeilenanfang beseitigen:
    !, tokenize([N|String],[T|Cs],Rs,Ts).
```

⟨*Trennzeichen und Zeilenumbruch beseitigen*⟩≡

```
tokenize([N|String],[T|Cs],Rs,Ts) :-
    worttrennung([T,N]),
    % Trennzeichen+Zeilenumbruch beseitigen:
    !, tokenize(String,Cs,Rs,Ts).
```

Ist das nächste Zeichen ein Zeilenende oder Leerzeichen, so bilden die gesammelten Zeichen ein Wort. Als Atom kommt es auf die Sammlung der Token, und man sucht den nächsten Wortanfang:

⟨*Erreichen der Tokengrenze*⟩≡

```
tokenize([T|String],R,Rs,Ts) :-
    zeilenende(T),
    % Sonstige Zeilenenden sind Tokengrenzen
    !, tokenToAtom(R,Atom),
    tokenize(String,[],[Atom|Rs],Ts).

tokenize([T|String],R,Rs,Ts) :-
    leerZeichen(T),
    % Sonst: Tokenende erreicht, neues suchen
    !, tokenToAtom(R,Atom),
    tokenize(String,[],[Atom|Rs],Ts).
```

Ist das nächste Zeichen ein Punkt, wird geprüft, ob er eine Abkürzung beendet und deshalb nicht das Satzende ist:

⟨*Punkt am Ende einer Suffixabkürzung*⟩≡

```
tokenize([S|String],R,Rs,Ts) :-
    punktZeichen(S),
    suffixAbkuerzung(Abkuerzung), %  Parkstr. 45
    reverse(Abkuerzung,RevAbk),
    append(RevAbk,_,[S|R]),
    !, tokenToAtom([S|R],Atom),
    tokenize(String,[],[Atom|Rs],Ts).
```

⟨*Punkt am Ende einer Abkürzung*⟩≡

```
tokenize([S|String],R,Rs,Ts) :-
    punktZeichen(S),
    abkuerzung(Abkuerzung),     %  z.T.
    append(Praefix,[S|Suffix],Abkuerzung),
    reverse(Praefix,R),
    append(Suffix,Rest,String),
    !, name(Atom,Abkuerzung),
    tokenize(Rest,[],[Atom|Rs],Ts).
```

Ist das nächste Zeichen ein Sonderzeichen, werden die gesammelten Zeichen als Wort aufgefaßt, aber ebenso das Sonderzeichen.

⟨*Sonderzeichen bilden eigene Token*⟩≡

```
tokenize([S|String],R,Rs,Ts) :-
    sonderzeichen(SAtom,S), % Sonderzeichen als Token
    !, tokenToAtom(R,Atom), % gelesene Zeichen als Token
    tokenize(String,[],[SAtom,Atom|Rs],Ts).
```

Man sammelt zwei Token und sucht den nächsten Wortanfang.

Ist das nächste Zeichen ein sonstiges Zeichen, wird es gesammelt und man liest weiter:

⟨*Erweitere Token*⟩≡

```
tokenize([C|String],R,Rs,Ts) :-
    % sonst: erweitere das angefangene Token
    !,tokenize(String,[C|R],Rs,Ts).
```

Gibt es kein weiteres Zeichen mehr, werden die gesammelten als Wort betrachtet und die umgekehrte Tokensammlung als Atomliste ausgegeben:

⟨*Eingabeende*⟩≡

```
tokenize([],[C|Cs],Rs,Ts) :-
    !, tokenToAtom([C|Cs],Atom),
    reverse([Atom|Rs],Ts).
tokenize([],[],Rs,Ts) :-
    reverse(Rs,Ts).
```

Hilfsprädikate des Tokenizers

⟨*Umwandlung der gesammelten Zeichen in ein Atom*⟩≡

```
tokenToAtom(RevToken,Atom) :-
    reverse(RevToken,Token),
    name(Atom,Token).
```

⟨*Datei- und Zeilenende, Worttrennung*⟩≡

```
dateiende(-1).            %  eof
zeilenende(10).           %  newline
worttrennung("-\n").      %  - newline
```

⟨*Satzendekriterien*⟩≡

```
satzende(". ").   %  nicht: .Char
satzende("? ").   satzende("! ").
satzende(": ").   satzende(".\n"). %  . newline
satzende("?\n"). satzende("!\n").
satzende(":\n").
```

⟨*Punkte am Abkürzungsende*⟩≡

```
abkuerzungsendeRev([Char,R|Seen]) :-
        leerZeichen(Char),punktZeichen(R),
        (abkuerzung(Abk); suffixAbkuerzung(Abk)),
        reverse(Abk,RevAbk),
        append(RevAbk,_,[R|Seen]).

suffixAbkuerzung("str.").
```

⟨*Abkürzungen: Monatsnamen*⟩≡

```
abkuerzung("Jan.").  abkuerzung("Feb.").
abkuerzung("Apr.").  abkuerzung("Aug.").
abkuerzung("Sept."). abkuerzung("Okt.").
abkuerzung("Nov.").  abkuerzung("Dez.").
```

⟨*Abkürzungen: Titel, Anreden*⟩≡

```
abkuerzung("Dr.").   abkuerzung("Prof.").
abkuerzung("Fa.").   abkuerzung("Frl.").
```

⟨*Sonstige Abkürzungen*⟩≡

```
abkuerzung("bzw.").  abkuerzung("etc.").
abkuerzung("usw.").  abkuerzung("vgl.").

abkuerzung("Nr.").   abkuerzung("a.a.O.").
abkuerzung("s.u.").  abkuerzung("m.M.").
abkuerzung("z.B.").  abkuerzung("z.T.").
```

⟨*Namensabkürzungen*⟩≡

```
abkuerzung([C,46]) :-  grossbuchstabe(C).
grossbuchstabe(C) :-
        member(C,"ABCDEFGHIJKLMNOPQRSTUVWXYZ").
```

⟨*Sonderzeichen*⟩≡

```
leerZeichen(32).          % name(' ',[32])

% Ausser in Zahlausdruecken und Abkuerzungen
% sollten folgende Zeichen eigene Token werden:
kommaZeichen(C) :- name(',',[C]).
punktZeichen(C) :- name('.',[C]).

sonderzeichen('.',46).  % name('.',[46]).  usw.
sonderzeichen(',',44).  sonderzeichen(';',59).
sonderzeichen(':',58).  sonderzeichen('!',33).
sonderzeichen('?',63).  sonderzeichen('"',34).  % bei Zit
sonderzeichen('(',40).  sonderzeichen(')',41).
```

⟨*Kriterium fuer Ordinalzahlende*⟩≡

```
ordinalzahlende([Ziffer,Punkt,Char]) :-
        ziffer(Ziffer),
        punktZeichen(Punkt),
        leerZeichen(Char).
ziffer(Z) :-
        member(Z,"0123456789").
```

Punkte wie in `der 3. Tag` gelten nicht als Satzenden.

Beispiel zur Tokenisierung

Prolog identifiziert einen String mit einer Liste von ASCII-Nummern

⟨*String zu Ascii*⟩≡

```
?- "Dr. Lee?" = L.
L = [68, 114, 46, 32, 76, 101, 101, 63]
```

Um den Trace lesbar zu machen, habe ich die Nummern durch entsprechende Zeichen ersetzt:

⟨*Anfrage*⟩≡

```
7 ?- ['tokenizer.pro'].
Yes
8 ?- trace.
Yes
9 ?- stringToAtoms("Dr. Lee?",Atome).

Call: (7) stringToAtoms([D,r,'.',' ',L,e,e,?],_G415) ?
```

⟨*Der erste Buchstabe wird untersucht und schließlich gesammelt*⟩≡

```
Call: (8) tokenize([D,r,'.',' ',L,e,e,?], [], [], _G41
Call: (9) leerZeichen(D) ?
Fail: (9) leerZeichen(D) ?
Call: (9) zeilenende(D) ?
Fail: (9) zeilenende(D) ?
Redo: (8) tokenize([D,r,'.',' ',L,e,e,?], [], [], _G41
Call: (9) sonderzeichen(_G414, D) ?
Fail: (9) sonderzeichen(_G414, D) ?
Redo: (8) tokenize([D,r,'.',' ',L,e,e,?], [], [], _G41
```

⟨*Dann wird getestet, ob die ersten beiden eine Trennung bilden*⟩≡

```
Call: (9) tokenize([r,'.',' ',L,e,e,?], [D], [], _G423
Call: (10) worttrennung([D,r]) ?
Fail: (10) worttrennung([D,r]) ?
Redo: (9) tokenize([r,'.',' ',L,e,e,?], [D], [], _G423
Call: (10) worttrennung([D,r]) ?
Fail: (10) worttrennung([D,r]) ?
Redo: (9) tokenize([r,'.',' ',L,e,e,?], [D], [], _G423
```

⟨*Der zweite Buchstabe wird untersucht und gesammelt*⟩≡

```
Call: (10) zeilenende(r) ?
Fail: (10) zeilenende(r) ?
Redo: (9) tokenize([r,'.',' ',L,e,e,?], [D], [], _G423
Call: (10) leerZeichen(r) ?
Fail: (10) leerZeichen(r) ?
Redo: (9) tokenize([r,'.',' ',L,e,e,?], [D], [], _G423
Call: (10) punktZeichen(r) ? s
Fail: (10) punktZeichen(r) ?
Redo: (9) tokenize([r,'.',' ',L,e,e,?], [D], [], _G423
Call: (10) punktZeichen(r) ? s
Fail: (10) punktZeichen(r) ?
Redo: (9) tokenize([r,'.',' ',L,e,e,?], [D], [], _G423
Call: (10) sonderzeichen(_G420,r) ?
Fail: (10) sonderzeichen(_G420, r) ?
Redo: (9) tokenize([r,'.',' ',L,e,e,?], [D], [], _G423
```

⟨*Bilden zweiter und dritter Buchstabe eine Worttrennung?*⟩≡

```
Call: (10) tokenize(['.',' ',L,e,e,?], [r,D], [], _G42
Call: (11) worttrennung([r,'.']) ?
Fail: (11) worttrennung([r,'.']) ?
Redo: (10) tokenize(['.',' ',L,e,e,?], [r,D], [], _G42
Call: (11) worttrennung([r,'.']) ?
Fail: (11) worttrennung([r,'.']) ?
Redo: (10) tokenize(['.',' ',L,e,e,?], [r,D], [], _G42
```

⟨*Der dritte Buchstabe wird untersucht und als Punkt erkannt*⟩≡

```
Call: (11) zeilenende('.') ?
Fail: (11) zeilenende('.') ?
Redo: (10) tokenize(['.',' ',L,e,e,?], [r,D], [], _G42
Call: (11) leerZeichen('.') ?
Fail: (11) leerZeichen('.') ?
Redo: (10) tokenize(['.',' ',L,e,e,?], [r,D], [], _G42
Call: (11) punktZeichen('.') ? s
Exit: (11) punktZeichen('.') ?
```

⟨*Bilden die gelesenen Zeichen eine Suffixabkürzung?*⟩≡

```
Call: (11) suffixAbkuerzung(_G429) ?
Exit: (11) suffixAbkuerzung([s,t,r,'.']) ?
Call: (11) reverse([s,t,r,'.'], _G445) ?
Exit: (11) reverse([s,t,r,'.'], ['.',r,t,s]) ?
Call: (11) append(['.',r,t,s], _G445, ['.',r,D]) ?
Fail: (11) append(['.',r,t,s], _G445, ['.',r,D]) ?
...
Redo: (10) tokenize(['.',' ',L,e,e,?], [r,D], [], _G42
Call: (11) punktZeichen('.') ? s
Exit: (11) punktZeichen('.') ?
```

⟨*Bilden die gelesenen Zeichen eine Abkürzung? Ja!*⟩≡

```
Call: (11) abkuerzung(_G429) ?
Exit: (11) abkuerzung([J,a,n,'.']) ?
...
Exit: (11) abkuerzung([D,r,'.']) ?
Call: (11) append(_G444, ['.'|_G435], [D,r,'.']) ?
Exit: (11) append([D,r], ['.'], [D,r,'.']) ?
Call: (11) reverse([D,r], [r,D]) ?
Exit: (11) reverse([D,r], [r,D]) ?
Call: (11) append([], _G457, [' ',L,e,e,?]) ?
Exit: (11) append([], [' ',L,e,e,?], [' ',L,e,e,?]) ?
Call: (11) name(_G456, [D,r,'.']) ?
Exit: (11) name('Dr.', [D,r,'.']) ?
```

⟨*Die Abkürzung wird als Token isoliert und das Folgezeichen untersucht*⟩≡

```
Call: (11) tokenize([' ',L,e,e,?], [], ['Dr.'], _G459)
Call: (12) leerZeichen(' ') ?
Exit: (12) leerZeichen(' ') ?
Call: (12) tokenize([L,e,e,?], [], ['Dr.'], _G459) ?
Call: (13) leerZeichen(L) ?
Fail: (13) leerZeichen(L) ?
Call: (13) zeilenende(L) ?
Fail: (13) zeilenende(L) ?
Redo: (12) tokenize([L,e,e,?], [], ['Dr.'], _G459) ?
Call: (13) sonderzeichen(_G456, L) ?
Fail: (13) sonderzeichen(_G456, L) ?
Redo: (12) tokenize([L,e,e,?], [], ['Dr.'], _G459) ?
```

⟨*Das Folgezeichen ist der Anfang eines neuen Tokens*⟩≡

```
Call: (13) tokenize([e,e,?], [L], ['Dr.'], _G465) ?
Call: (14) worttrennung([L,e]) ?
Fail: (14) worttrennung([L,e]) ?
Redo: (13) tokenize([e,e,?], [L], ['Dr.'], _G465) ?
Call: (14) worttrennung([L,e]) ?
Fail: (14) worttrennung([L,e]) ?
Redo: (13) tokenize([e,e,?], [L], ['Dr.'], _G465) ?
Call: (14) zeilenende(e) ?
Fail: (14) zeilenende(e) ?
Redo: (13) tokenize([e,e,?], [L], ['Dr.'], _G465) ?
```

⟨*Das nächste Zeichen wird untersucht ...* ⟩≡

```
Call: (14) leerZeichen(e) ?
Fail: (14) leerZeichen(e) ?
Redo: (13) tokenize([e,e,?], [L], ['Dr.'], _G465) ?
Call: (14) punktZeichen(e) ? s
Fail: (14) punktZeichen(e) ?
Redo: (13) tokenize([e,e,?], [L], ['Dr.'], _G465) ?
Call: (14) punktZeichen(e) ? s
Fail: (14) punktZeichen(e) ?
Redo: (13) tokenize([e,e,?], [L], ['Dr.'], _G465) ?
Call: (14) sonderzeichen(_G602, e) ?
Fail: (14) sonderzeichen(_G602, e) ?
Redo: (13) tokenize([e,e,?], [L], ['Dr.'], _G465) ?
```

⟨*.. und setzt das angefangene Wort fort ..*⟩≡

```
Call: (14) tokenize([e,?], [e,L], ['Dr.'], _G471) ?
Call: (15) worttrennung([e,e]) ?
Fail: (15) worttrennung([e,e]) ?
Redo: (14) tokenize([e,?], [e,L], ['Dr.'], _G471) ?
Call: (15) worttrennung([e,e]) ?
Fail: (15) worttrennung([e,e]) ?
Redo: (14) tokenize([e,?], [e,L], ['Dr.'], _G471) ?
Call: (15) zeilenende(e) ?
Fail: (15) zeilenende(e) ?
```

⟨*Das nächste Zeichen wird untersucht ...* ⟩+≡

```
Redo: (14) tokenize([e,?], [e,L], ['Dr.'], _G471) ?
Call: (15) leerZeichen(e) ?
Fail: (15) leerZeichen(e) ?
Redo: (14) tokenize([e,?], [e,L], ['Dr.'], _G471) ?
Call: (15) punktZeichen(e) ? s
Fail: (15) punktZeichen(e) ?
Redo: (14) tokenize([e,?], [e,L], ['Dr.'], _G471) ?
Call: (15) punktZeichen(e) ? s
Fail: (15) punktZeichen(e) ?
Redo: (14) tokenize([e,?], [e,L], ['Dr.'], _G471) ?
Call: (15) sonderzeichen(_G461, e) ?
Fail: (15) sonderzeichen(_G461, e) ?
Redo: (14) tokenize([e,?], [e,L], ['Dr.'], _G471) ?
```

⟨*und setzt das angefangene Wort fort*⟩≡

```
Call: (15) tokenize([?], [e,e,L], ['Dr.'], _G477) ?
Call: (16) worttrennung([e,?]) ?
Fail: (16) worttrennung([e,?]) ?
Redo: (15) tokenize([?], [e,e,L], ['Dr.'], _G477) ?
```

⟨*Das nächste Zeichen wird untersucht ...* ⟩+≡

```
Call: (16) zeilenende(?) ?
Fail: (16) zeilenende(?) ?
Redo: (15) tokenize([?], [e,e,L], ['Dr.'], _G477) ?
Call: (16) leerZeichen(?) ?
Fail: (16) leerZeichen(?) ?
Redo: (15) tokenize([?], [e,e,L], ['Dr.'], _G477) ?
Call: (16) punktZeichen(?) ? s
Fail: (16) punktZeichen(?) ?
Redo: (15) tokenize([?], [e,e,L], ['Dr.'], _G477) ?
Call: (16) punktZeichen(?) ? s
Fail: (16) punktZeichen(?) ?
Redo: (15) tokenize([?], [e,e,L], ['Dr.'], _G477) ?
```

⟨*und ist ein Sonderzeichen, das nicht zum angefangenen Token gehört*⟩≡

```
Call: (16) sonderzeichen(_G474, ?) ?
Exit: (16) sonderzeichen(?, ?) ?

Call: (16) tokenToAtom([e,e,L], _G475) ?
Call: (17) reverse([e,e,L], _G478) ?
Exit: (17) reverse([e,e,L], [L,e,e]) ?
Call: (17) name(_G486, [L,e,e]) ?
Exit: (17) name('Lee', [L,e,e]) ?
Exit: (16) tokenToAtom([e,e,L], 'Lee') ?
```

⟨*sodaß zwei weitere Token erkannt sind*⟩≡

```
Call: (16) tokenize([], [], [?,'Lee','Dr.'], _G492) ?
```

Da die Eingabe keine weiteren Zeichen enthält, wird die Antwort gebildet:

⟨*Die Tokensammlung wird invertiert und bildet das Ergebnis*⟩≡

```
    Call: (17) reverse([?,'Lee','Dr.'], _G493) ?
    Exit: (17) reverse([?,'Lee','Dr.'],
                                ['Dr.', 'Lee', ?]) ?
    Exit: (16) tokenize([], [], [?, 'Lee', 'Dr.'],
                                ['Dr.', 'Lee', ?]) ?
    Exit: (15) tokenize([?], [e,e,L], ['Dr.'],
                                ['Dr.', 'Lee', ?]) ?
    ...
    Exit: (8) tokenize([D,r,'.',' ',L,e,e,?], [], [],
                          ['Dr.', 'Lee', ?]) ?
    Exit: (7) stringToAtoms([D,r,'.',' ',L,e,e,?],
                          ['Dr.', 'Lee', ?]) ?
  Yes
```

Horn-Klausel-Grammatiken (DCG)

Differenzlisten

Eine Teilfolge $[a_{i+1}, \ldots, a_j]$ einer Folge

$$L = [a_1, \ldots, a_{i+1}, \ldots, a_{j+1}, \ldots a_n]$$

kann als *Differenz* $I - J$ der Suffixe $I = [a_{i+1}, \ldots, a_n]$ und $J = [a_{j+1}, \ldots, a_n]$ dargestellt werden.

Beachte: für jede Liste L und Elemente $a_1, \ldots, a_n$ gilt:

- $[a_1, \ldots a_n] = [a_1, \ldots, a_n | L] - L$
- $L = L - []$
- $[] = L - L$

Aber diese Gleichungen sind nicht Teil der Unifikation.

Um festzustellen, ob L eine Teilfolge X mit $p(X)$ hat, d.h.

$$\exists i, j\,(0 \leq i \leq j \leq n \;\wedge\; p([a_{i+1}, \ldots, a_j])),$$

testet man $p(I - J)$, wobei man I, J beim Durchlaufen von $L = [a_1, \ldots, a_i | I]$ und $I = [a_{i+1}, \ldots, a_j | J]$ erhält.

Oft spart man sich das Funktionszeichen $-$, indem man `p/1` in ein Prädikat `p/2` übersetzt und dann $p(I, J)$ benutzt.

Append von Differenzlisten in O(1)

Wir hatten gesehen, daß die Berechnung der Listenverkettung

$$[a_{i+1}, \ldots, a_j] \ + \ [b_{j+1}, \ldots, b_k] = [a_{i+1}, \ldots, b_k]$$

durch `append/3` soviele Schritte braucht, wie die Summe der Längen der Eingabelisten, $(j - i) + (k - j) = k - i$.

In der Darstellung mit Differenzlisten besagt die Verkettung:

$$\begin{aligned} &([a_{i+1}, \ldots, a_j | Bs] - Bs) \ + \ ([b_{j+1}, \ldots, b_k | Cs] - Cs) \\ &\quad = \ [a_{i+1}, \ldots, a_j, b_{j+1}, \ldots, b_k | Ds] - Ds \end{aligned}$$

für *beliebige* Listen Bs, Cs, Ds.

Der Nutzen von Differenzlisten beruht darauf, daß man für die *spezielle* Wahl

$$\underbrace{[a_{i+1}, \ldots, a_j | Bs]}_{As} - Bs \;+\; \underbrace{[b_{j+1}, \ldots, b_k | Cs]}_{Bs} - Cs$$
$$= \underbrace{[a_{i+1}, \ldots, a_j, b_{j+1}, \ldots, b_k | Cs]}_{As} - Cs$$

die Ergebnisliste $As - Cs$ in $O(1)$ Schritten berechnen kann: von den 4 Eingabelisten $As - Bs, Bs - Cs$ gibt man 2 zurück.

$$(As - Bs) + (Bs - Cs) = (As - Cs).$$

Die Verkettung von Differenzlisten wird also durch

⟨*Differenzlistenappend*⟩≡

```
appdl(As-Bs,Bs-Cs,As-Cs).     (*)
```

definiert. Da (außer Unifikation) keine Auswertung im Spiel ist, geht das in *konstantem* statt in linearem Aufwand.

Daher benutzt man die Differenzlistendarstellung, um Anwendungen von `append` zu vermeiden: Aufrufe der Form `(*)` kann man weglassen.

Beispiel: Differenzlisten und Akkumulatoren

Differenzlisten sind oft nur eine andere Schreibweise für die Verwendung von Akkumulatorvariablen.

⟨*Reverse mit Differenzlistendarstellung der Ausgabe*⟩≡

```
reverse(Xs,Ys) :-
        revdl(Xs,Ys-[]).

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

⟨*Reverse mit zusätzlichem Akkumulator*⟩≡

```
reverse(Xs,Ys) :-
        rev(Xs,Ys,[]).

rev([X|Xs],Ys,Acc) :-
        rev(Xs,Ys,[X|Acc]).
rev([],Acc,Acc).
```

Hier wird wirklich nur $p(X, I - J)$ durch $p(X, I, J)$ ersetzt.

Kontextfreie Grammatiken (CFG) in Prolog

⟨*Beispiel einer kontextfreien Grammatik G*⟩≡

```
s -> np vp
np -> det n
vp -> v np | v

det -> der | das
n -> Programmierer | Programm
v -> steht | schrieb
```

```
Erzeugung von Sätzen:
s -> np vp
  -> det n v
  -> das Programm steht
```

Eine Grammatik definiert Eigenschaften von Wortfolgen. Für eine Implementierung in Prolog liegt daher nahe:

- Wörter sind Prolog-Atome
- Ausdrücke sind Wortfolgen = Listen von Prolog-Atomen
- Kategorien sind Eigenschaften von Wortfolgen = Prädikate

Direkte Übersetzung in Prolog

⟨*Grammatik G als Menge von Prolog-Prädikaten*⟩≡

```
s(S) :- append(NP,VP,S), % Zerteile S. Haben
        np(NP),          % die Teilfolgen die
        vp(VP).          % Eigenschaften np,vp?
np(NP) :- append(D,N,NP),
          det(D), n(N).
vp(VP) :- append(V,NP,VP),
          v(V), np(NP).
vp(VP) :- v(VP).

det([der]). det([das]).
n(['Programmierer']). n(['Programm']).
v([steht]). v([schrieb]).
```

Parsen mit Prolog

Die Analyse von Eingaben ist damit möglich:

⟨*Trace einer Analyse*⟩≡

```
?- trace.

?- np([das,'Programm']).
Call:( 8) np([das,Programm]) ?
Call:( 9) append(_L128,_L129,[das,Programm]) ?
Exit:( 9) append([],[das,Programm],[das,Programm])?
Call:( 9) det([]) ?
Fail:( 9) det([]) ?
Redo:( 9) append(_L128,_L129,[das,Programm]) ?
Call:(10) append(_G308,_L129,[Programm]) ?
Exit:(10) append([],[Programm],[Programm]) ?
Exit:( 9) append([das],[Programm],[das,Programm])?
```

⟨*Trace einer Analyse*⟩+≡

```
Call:( 9) det([das]) ?
Exit:( 9) det([das]) ?
Call:( 9) n([Programm]) ?
Exit:( 9) n([Programm]) ?
Exit:( 8) np([das,Programm]) ?
```

Aber: Das wird zu langsam, da bei großen Beispielen zu viel Zeit und Platz von `append` verbraucht wird.

Da `append(Xs,Ys,Zs)` gleichbedeutend mit

```
Xs = Zs-Ys, appdl(Zs-Ys,Ys-[],Zs-[])
```

und hierin der zweite Teil immer wahr ist, kann man die Verwendung von `append` vermeiden, indem man im Programm `Xs` als Differenz `Zs-Ys` darstellt. (`Xs=Zs-Ys` ist nicht Unifikation!)

Neue Übersetzung von syntaktischen Regeln

Stellt man in der direkten Übersetzung der ersten CFG-Regel, d.h.

⟨*Übersetzung mit einstelligen Prädikaten über Listen*⟩≡

```
s(S) :- append(NP,VP,S),
        np(NP),
        vp(VP).
```

die Listen `S` durch `S-[]`, `NP` durch `S-VP` und `VP` durch `VP-[]` dar, so erhält man

⟨*Übersetzung mit Prädikaten über Differenzlisten*⟩≡

```
s(S-[]) :-
        appdl(S-VP,VP-[],S-[]),
        np(S-VP),
        vp(VP-[]).
```

worin der `appdl`-Aufruf entfallen kann!

Statt `I-J` schreibt man *zwei* Argumente (für Suffixlisten, also Positionen i, j in der Eingabeliste S), und hat

⟨*Übersetzung mit zweistelligen Prädikaten über Suffixlisten*⟩≡

```
s(S) :- s(S,[]).
s(I,J) :-            % von I bis J ein s, wenn
        np(I,K),     % von I bis K eine np und
        vp(K,J).     % von K bis J eine vp
```

Auch das wird oft als Differenzlistendarstellung bezeichnet.

Neue Übersetzung von lexikalischen Regeln

Bei lexikalischen CFG-Regeln wie

```
det -> der.
```

erhält man aus der Übersetzung mit Prädikat über Wortlisten

```
det([der]).      bzw.      det(L) :- L = [der].
```

folgende Übersetzung mit zweistelligem Prädikat über Suffixlisten:

⟨*Übersetzung mit Differenzlisten*⟩≡

```
det([der|J],J).   bzw.     det(I,J) :- I = [der|J].
```

Ähnlich behandelt man Wörter in syntaktischen Regeln:

Ein Funktionsverbgefüge wie

⟨*Verbgefüge*⟩≡

```
  fvp -> bringt np zum Platzen
```

wird übersetzt zu

⟨*Übersetzung mit Differenzlisten*⟩+≡

```
  fvp(I,J) :-
          I = [bringt|K],
          np(K,L),
          L = [zum,'Platzen'|J].
```

Neu übersetzte Grammatik

Insgesamt erhält man so eine neue

⟨*Übersetzung der Grammatik G*⟩≡

```
parse(S) :- s(S,[]).

s(I,J)   :- np(I,K), vp(K,J).
np(I,J)  :- det(I,K), n(K,J).
vp(I,J)  :- v(I,K), np(K,J).
vp(I,J)  :- v(I,J).

det([der|J],J).
det([das|J],J).
n(['Programmierer'|J],J).
n(['Programm'|J],J).
v([steht|J],J).
v([schrieb|J],J).
```

Parsen mit neu übersetzter Grammatik

Beim Parsen treten keine `append`-Aufrufe mehr auf:

⟨*Parse*⟩≡

```
?- trace.
?- np([das,'Programm'],[]).
   Call:(8) np([das,Programm],[]) ?
   Call:(9) det([das,Programm],_L129) ?
   Exit:(9) det([das,Programm],[Programm]) ?
   Call:(9) n([Programm],[]) ?
   Exit:(9) n([Programm],[]) ?
   Exit:(8) np([das,Programm],[]) ?
Yes
```

Aber: Bei linksrekursiven Regeln wie `np -> np und np` gerät der Parser in eine Endlosschleife.

Format der Klauselgrammatiken (DCG)

Eine solche Übersetzung von Grammatikregeln in Klauseln macht Prolog automatisch, wenn man die CFG im DCG-Format schreibt:

- Grammatikregeln werden mit `-->` (statt `->`) geschrieben.
- Konstituenten werden durch `,` getrennt.
- Kategorien können Terme, nicht bloß Atome, sein, und daher z.B. auch Merkmale enthalten, wie `np(Numerus,Kasus)`.
- Terminale (Wörter) *a* sind als Atomliste `[a]` zu schreiben.
- Regeln dürfen Nebenbedingungen `{Bedingung}` enthalten.
- Alternativen werden mit `;` (statt `|`) geschrieben.

Eine *syntaktische* DCG-Regel wie

```
p(X) --> q(X), {r(X)}, s(X), [a], t(X).
```

wird übersetzt in die Prolog-Klausel

```
p(X,I,J) :- q(X,I,K), r(X), s(X,K,L),
            L = [a|M], t(X,M,J).
```

Eine *lexikalische* Regel wie

```
p(X) --> [Atom].
```

wird übersetzt in ein Faktum

```
p(X,[Atom|J],J).
```

Die Differenzlistenargumente werden *nach* den Argumenten der Kategorienprädikate angefügt.

Beispielgrammatik als DCG

⟨*programmierer.pl*⟩≡

```
s --> np, vp.
np --> det, n.
vp --> v, np.
vp --> v.

det --> [der].
det --> [das].
n --> ['Programmierer'].
n --> ['Programm'].
v --> [steht].
v --> [schrieb].
```

Format der Klauselgrammatiken (DCG)

Allgemein kann eine DCG-Regel wie folgt aufgebaut werden:

⟨*DCG-Regel Aufbau*⟩≡

```
Grammatikregel := Links --> Rechts.

Links := Cat
Rechts := Cat
       | (Rechts, Rechts)  % Abfolge
       | (Rechts; Rechts)  % Alternative
       | []                % leeres Wort
       | [Atom|Atome]      % Terminalsymbole
       | { Prolog-Code }   % Nebenbedingung

Cat := Term  % Kategorienname mit oder
             % ohne Argumenten (keine Variable,
             % keine Prolog-Konstante)
```

Beachte, daß man auf der rechten Seite einer Regel Abfolgen und Alternativen schachteln darf, z.B.

⟨*Schachtelung*⟩≡

```
np(Gen,Num,Kas) -->
    art(Gen,Num,Kas),
    % optionale Adjektivphrase:
    (ap(Gen,Num,Kas) ; []),
    n(Gen,Num,Kas),
    { genus(Gen), numerus(Num), kasus(Kas) }.
```

Eine DCG-Datei ist eine Prolog-Programmdatei und kann auch normale Prolog-Klauseln enthalten, z.B.

⟨*Merkmaldefinition* ⟩≡

```
genus(fem). genus(mask). genus(neutr).
```

Ausgabe der syntaktischen Analyse

Eine *syntaktische Analyse* beschreibt den Aufbau eines Ausdrucks gemäß einer Grammatik. Wir verwenden dazu Bäume, deren Knoten mit Wörtern und Ausdrucksarten markiert sind.

Bei einer DCG ist die Analyse ein Prolog-Term. Die Kategorien der Grammatik erhalten dafür ein Argument, z.B.

⟨*Baumaufbau in Nebenbedingungen*⟩≡

```
np(Baum) -->
    n(Baum1), [und], np(Baum2),
    { Baum = np(Baum1, und, Baum2) }.
```

In { ... } wird gesagt, wie der Syntaxbaum aus den Analysen der Teilausdrücke zusammengesetzt wird.

Ebensogut kann man die Struktur des Baums auch im Kopf der Regel angeben, d.h. durch

⟨*Baumaufbau im Kopf der Regel*⟩≡

```
np(np(Baum1, und, Baum2)) -->
    n(Baum1), [und], np(Baum2).
```

Bei der Übersetzung der Grammatikregel in eine Klausel werden die Differenzlistenargumente hinzugefügt:

⟨*Übersetzung bei Baumaufbau in Nebenbedingungen*⟩≡

```
np(Baum,I,J) :-
    n(Baum1,I,K), K = [und|L], np(Baum2,L,J),
    Baum = np(Baum1,und,Baum2).
```

bzw.

⟨*Übersetzung bei Baumaufbau im Kopf der Regel*⟩≡

```
np(np(Baum1,und,Baum2),I,J) :-
    n(Baum1,I,K), K = [und|L], np(Baum2,L,J).
```

Beispielgrammatik
mit Ausgabe der Analyse

Einige Regeln im ersten Format:

⟨*Baumargument in den Kategorien*⟩≡

```
s(S)     --> np(NP), vp(VP), ['.'],
             {S = s(NP, VP)}.
np(NP)   --> det(Det), n(N),
             {NP = np(Det,N)}.

fvg(FVG) --> [bringt], np(NP), [zum,'Absturz'],
             {FVG = vp(fv(bringt_zum_Absturz),NP)}.

det(Det) --> [der], {Det = det(der)}.
det(Det) --> [das], {Det = det(das)}.
```

... und einige im zweiten Format:

⟨*Baumargument im Kopf der Regel*⟩≡

```
vp(FVG) --> fvg(FVG).
vp(vp(vtr(V),NP)) --> v(vtr(V)), np(NP).
vp(vp(vitr(V))) --> v(vitr(V)).

n(n('Programmierer')) --> ['Programmierer'].
n(n('Programm')) --> ['Programm'].

v(vitr(steht)) --> [steht].
v(vtr(schrieb)) --> [schrieb].
v(vitr(schrieb)) --> [schrieb].
```

Beispielgrammatik: Parse-Aufrufe

Zur einfacheren Bedienung verpacken wir den Aufruf zur Analyse von Ausdrücken verschiedener Arten in ein gemeinsames `parse/2`:

⟨*Parseaufrufe*⟩≡

```
parse(Atome,Baum) :-   s(Baum,Atome,[]).
parse(Atome,Baum) :-  vp(Baum,Atome,[]).
parse(Atome,Baum) :-   v(Baum,Atome,[]).
parse(Atome,Baum) :-  np(Baum,Atome,[]).
parse(Atome,Baum) :-   n(Baum,Atome,[]).
parse(Atome,Baum) :- det(Baum,Atome,[]).
```

⟨*programmierer.baum.pl*⟩≡

⟨*Baumargument in den Kategorien*⟩
⟨*Baumargument im Kopf der Regel*⟩
⟨*Parseaufrufe*⟩

⟨*Syntaxanalyse mit Baumausgabe*⟩≡

```
?- ['programmierer.baum.pl'].

?- parse([das,'Programm'],Baum).
Baum = np(det(das), n('Programm'))
?- parse([schrieb,das,'Programm'],Baum).
Baum = vp(vtr(schrieb),
          np(det(das), n('Programm')))

?- parse([der,'Programmierer',schrieb,'.'],Baum).
Baum = s(np(det(der), n('Programmierer')),
         vp(vitr(schrieb)))
?- parse([der,'Programmierer',bringt,das,'Programm',
          zum,'Absturz','.'],Baum).
Baum = s(np(det(der), n('Programmierer')),
         vp(fv(bringt_zum_Absturz),
            np(det(das), n('Programm'))))
```

Erweiterung um syntaktische Merkmale

Zur Überprüfung von Rektions- und Kongruenzbedingungen erweitern wir die Kategorien um ein weiteres Argument, eine Liste von Merkmalen:

⟨*Kategorie mit Merkmalslisten (und Baumausgabe)*⟩≡

```
np([Kas],NP) -->
      det([GenusD,KasD],Det), n([GenusN,KasN],N),
      { GenusD = GenusN
      -> (KasD = KasN
          -> Kas = KasN, NP = np(Det,N)
          ; write('Kongruenzfehler im Kasus.\n'),
            fail)
      ; write('Kongruenzfehler im Genus.\n'),
          fail }.
```

Terminierung mit linksrekursiven Regeln

Linksrekursive Regeln wie

```
np(NP) --> np(NP1), [und], np(NP2),
           { NP = np(NP1,und,NP2) }.
```

führen zu einer endlosen Beweissuche. Beschränkt man die Rekursionstiefe der linken Konstituente, so terminiert sie:

⟨*Beschränkung der Rekursionstiefe*⟩≡

```
np(NP) --> { Tiefe = 0 ; Tiefe = 1 },
           np(Tiefe,NP).
np(Tiefe,NP) --> { T is Tiefe-1 , 0 =< T },
             np(T,NP1), [und], np(NP2),
             { NP = np(NP1,und,NP2) }.

np(0,NP) --> det(Det), n(N), {NP = np(Det,N)}.
```

Bem. Die Stelligkeit von `np` in `parse(Atome,Baum)` anpassen!

Darstellung lexikalischer Regeln

Es ist lästig, daß man bei lexikalischen Regeln, bei denen man das jeweilige Wort als Analyse ausgeben möchte, dieses Wort zweimal schreiben muß, wie in

⟨*Beispielregel*⟩≡

```
  n([neut,nom,sg],Baum) -->
        ['Programm'], { Baum = n('Programm') }.
```

oder in

⟨*Beispielregel*⟩+≡

```
  n([neut,nom,sg],n('Programm')) --> ['Programm'].
```

Um diese Doppelarbeit (und damit verbundenen Fehler) zu vermeiden, schreibt man Lexikoneinträge besser in der Form

⟨*Lexikoneintrag als Faktum*⟩≡

```
n([neut,nom,sg],'Programm').
```

und ergänzt die Grammatik um *eine* zusätzliche Regel:

⟨*Lexikonzugriff in einer Nebenbedingung*⟩≡

```
n(Merkmale,Baum) --> [Wort],
        { n(Merkmale,Wort), Baum = n(Wort) }.
```

oder gleich um die entsprechende Prolog-Klausel:

⟨*Lexikonzugriff als Bedingung einer Klausel*⟩≡

```
n(Merkmale,n(Wort),[Wort|J],J) :-
        n(Merkmale,Wort).
```

Oft verwendet man stattdessen Einträge der Form

⟨*Lexikoneinträge*⟩≡

```
lexicon(Wort,cat(Merkmale)).
```

und eine entsprechende Klausel pro Kategorie,

⟨*Klausel für jeweilige Kategorie*⟩≡

```
cat(Merkmale,cat(Wort),[Wort|J],J) :-
      lexicon(Wort,cat(Merkmale)).
```

Übersetzung von DCG-Regeln in Klauseln

Die Umwandlung von DCG-Regeln in Prolog-Klauseln ist die Übersetzung eines Quell- in ein Zielprogramms.

Solche Übersetzungen sind in Prolog besonders einfach zu verwirklichen, denn

- Prolog-Klauseln sind spezielle Prolog-Terme, sodaß Programme als Daten betrachtet werden können.
- Prolog-Terme lassen sich mit eingebauten Prädikaten analysieren, zerlegen und zusammensetzen.
- Prolog erlaubt es, die in die Prolog-Datenbank geladenen Klauseln durch ein laufendes Programm zu verändern.

Die Veränderung der Prolog-Datenbank brauchen wir noch nicht.

Beispiele:

- Klauseln sind Terme: `:-/2` ist ein infix geschriebener Operator

  ⟨*Unifikation mit Klauseln*⟩≡

  ```
  ?- (Kopf :- Rumpf) = (p(X,Y) :- q(X,Y),! ; r(X,Y)).
  Kopf = p(X, Y),
  Rumpf = (q(X, Y), !;r(X, Y)).
  ```

- Termuntersuchung, -zerlegung, -zusammensetzung:

  ⟨*Termart testen*⟩≡

  ```
  var/1, atom/1, atomic/1, ground/1, compound/1,
  ```

  ⟨*Term zerlegen*⟩≡

  ```
  functor(?Term,?Functor,?Arity)
  arg(?ArgNumber,+Term,?ArgTerm)
  +Term =.. [?Functor | ?ArgTerms]
  ```

  ⟨*Term zusammensetzen*⟩≡

  ```
  -Term =.. [+Funktion | +Argumentterme]
  ```

Programme übersetzen: DCG $\mapsto$ Prolog

	DCG	Prolog
Nonterminal:	`f(A1,..,An)`	`f(A1,..,An,I,J)`
Terminale:	`[Atom1,...]`	`I = [Atom1,...|J]`
Bedingung:	`{Prolog}`	`Prolog`
Regel:	`L --> R.`	`tr(L,I,J) :- tr(R,I,J).`

⟨*translate.pl*⟩≡

```
translate((Links --> Rechts),(Kopf :- Rumpf)) :-
    !, translate(Links,Kopf,I,J),
    translate(Rechts,Rumpf,I,J).
translate(Term,Term). % Hilfsprädikate nicht ändern

% translate(+Rumpf,-RumpfTr,+Von,-Bis)
translate((A,B),(ATr,BTr),I,J) :-
    !, translate(A,ATr,I,K),
    translate(B,BTr,K,J).
```

⟨*translate.pl*⟩+≡

```
translate((A;B),(ATr;BTr),I,J) :-
    !, translate(A,ATr,I,J),
    translate(B,BTr,I,J).
translate([A|As],(I = Liste),I,J) :-
    !, terminals([A|As]),
    append([A|As],J,Liste).
translate([],(I=J),I,J) :- !.
translate({Prolog},(Prolog,I=J),I,J) :- !.
translate(A,ATr,I,J) :-
    nonterminal(A),
    !, A =.. [F|Args],
    append(Args,[I,J],ArgsIJ),
    ATr =.. [F|ArgsIJ].
nonterminal(A) :- atom(A) ; compound(A).
terminals([A|As]) :- atom(A),terminals(As).
terminals([]).
```

Beispielübersetzungen

Das Prädikat `portray_clause/1` stellt eine Klausel mit lesbaren Variablennamen und Einrückung des Klauselkörpers dar. Lassen wir uns damit

```
?- translate((np --> det,n,(np;pp)), Klausel),
   portray_clause(Klausel).
```

die übersetzte Klausel anzeigen, so erhalten wir:

```
np(A, B) :-
    det(A, C),
    n(C, D),
    (   np(D, B)
    ;   pp(D, B)
    ).
```

Jede Kategorie hat zwei neue Argumente `I`, `J`, zur Darstellung der (impliziten) `Eingabe` als Differenzliste `I = [Eingabe|J]`.

Entsprechend bei Kategorien mit Merkmalen, aber nicht bei den Bedingungen in `{...}`. Aus der Grammatikregel

```
?- translate((np(Kas) --> det(Kas),n(Kas),np(gen),
                {member(Kas,[nom,gen,dat,akk])}),
               Klausel),
   portray_clause(Klausel).
```

erhalten wir die Klausel:

```
np(A, B, C) :-
    det(A, B, D),
    n(A, D, E),
    np(gen, E, C),
    member(A, [nom, gen, dat, akk]).
```

Die Nebenbedingung erhält keine Differenzlistenargumente.

Übersetze eine DCG-Datei in eine Prolog-Datei

Eine DC-Grammatik liege in einer `Eingabedatei` vor.

Sie soll übersetzt und in `Ausgabedatei` geschrieben werden.

⟨*translate.pl*⟩+≡

```
uebersetze(Eingabedatei,Ausgabedatei) :-
    open(Eingabedatei,read,Eingabestrom),
    open(Ausgabedatei,write,Ausgabestrom),
    uebersetzeRegeln(Eingabestrom,
                     Ausgabestrom),
    close(Eingabestrom),
    close(Ausgabestrom).
```

⟨*translate.pl*⟩+≡

```
uebersetzeRegeln(Eingabestrom,Ausgabestrom) :-
    read(Eingabestrom,Regel), % kuerzt Strom!
    (Regel = end_of_file ->
        write(user,'\nFertig. \n')
    ; translate(Regel,Klausel),
      write(user,'\n\nRegel: '),
      write(user,Regel),
      write(user,'\nÜbersetzung: \n'),
      portray_clause(Klausel),
      portray_clause(Ausgabestrom,Klausel),
      uebersetzeRegeln(Eingabestrom,
                       Ausgabestrom)).
```

Einlesen der Eingabe ...

Da es mühsam ist, Beispielsätze als Listen von Atomen einzugeben, lassen wir den Benutzer einen Beispielsatz eintippen und wandeln ihn automatisch in eine Liste von Atomen um:

```
satzende :-
    write('Beende die Eingabe mit <Punkt><Return>'),
    nl,read_sentence(user,Sentence,[]),
    name(Satz,Sentence),
    nl,write('Eingabe:   '),writeq(Satz),
    tokenize(Sentence,Atomlist),
    nl,write('Atomliste: '),
    writeq(user,Atomlist).
```

Die Eingabe lesen wir zeichenweise bis zum Satzende und geben sie (als Liste `Sentence` von ASCII-Nummern) aus; `Seen` ist ein Akkumulator, der die schon gelesenen Zeichen sammelt:

```
read_sentence(Stream,Sentence,Seen) :-
    get0(Stream,Char),
    !,read_sentence(Stream,Sentence,Char,Seen).

read_sentence(Stream,Sentence,Char,[C|Seen]) :-
    satzende([C,Char]),
    !,reverse([C|Seen],Sentence).
read_sentence(Stream,Sentence,Char,Seen) :-
    read_sentence(Stream,Sentence,[Char|Seen]).
```

Als Satz- bzw. Eingabeende gelte “.<Return>“:

```
satzende([46,10]). % <Punkt><Return>
```

... und Zerlegung in Wörter (Tokenisierung)

Schließlich wandeln wir die Liste von ASCII-Nummern in ein Atom um und spalten dieses an den Leerzeichen in einzelne Atome auf:

```
tokenize(String,Atomlist) :-
        name(Atom,String),
        concat_atom(Atomlist,' ',Atom).
```

Graphische Anzeige der Analyse

Mit dem Graphikprogramm `dot` zeigen wir die Analyse an:

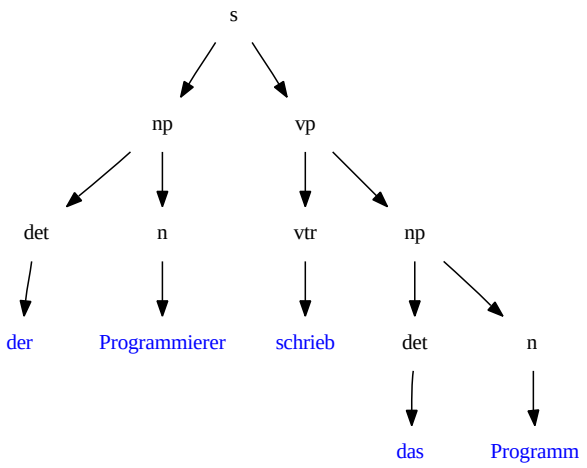
⟨*parse.pl*⟩≡

```
parse :-
    ensure_loaded('tokenizer.pl'),
    ensure_loaded('dottree.pl'),
    write(user,'Gib einen Ausdruck ein!\n'),
    read_sentence(user,Sentence,[]),
    stringToAtoms(Sentence,Atomlist),
    (parse(Atomlist,Baum)
    -> write(user,'Die Analyse ist: '),
        displayTree(Baum)
    ; write(user,'Das war nicht korrekt. ')).
```

Zu `displayTree/1` siehe die Übungsstunde.

⟨*Parsen mit graphischer Baumanzeige*⟩≡

```
?- ['programmierer.baum',parse].
?- parse.
|: der Programmierer schrieb das Programm.
Die Analyse ist:
```



Partielle Auswertung

Ein *partieller Auswerter* berechnet aus einem *Quellprogramm* und (einigen von) dessen *Argumenten* ein *Zielprogramm*, das ausgibt, was das Quellprogramm für diese Argumente ausgäbe.

- Gegeben: Quellprogramm `p(+X,+Y,-Z)` und Wert `t` für `X`.
- Ausgabe: Zielprogramm $\mathtt{p}_t$`(+Y,-Z)`, das `t` nicht mehr betrachtet, aber dieselben Ausgaben wie `p(t,+Y,-Z)` liefert.

Sind die Argumente des Quellprogramms teilweise gegeben, kann die Berechnung des Quellprogramms teilweise ausgeführt werden.

Beispiel: universeller Parser $\mapsto$ Parser zu Grammatik G

- `parse(+Grammatik,+Eingabe,-Analyse)`
- `parser(+Eingabe,-Analyse)` für eine bestimmte Grammatik

Prädikate zur Manipulation der Datenbank

⟨*Klauseln in die Datenbank schreiben*⟩≡

```
?- assert(+Term), assertz(+Term).
   z.B. assert((Kopf :- Rumpf)).
```

⟨*Klauseln in der Datenbank löschen*⟩≡

```
?- retract(+Term), retractall(Kopf),
   z.B. retractall(p(X,Y)).
```

⟨*Prädikat in der Datenbank löschen*⟩≡

```
?- abolish(+Functor/Arity),    z.B. abolish(p/2).
```

⟨*Klausel in der Datenbank suchen*⟩≡

```
?- clause(+Head,-Body).
```

⟨*Prüfen, aus welcher Datei das Prädikat eines Kopfs kommt*⟩≡

```
?- predicate_property(Module:Head,file(Filename)).
```

Das Quellprogramm, die zu eliminierenden Prädikate/Köpfe und die Definitionen für diese Prädikate/Köpfe seien so angegeben:

```
% Quellprogramm:
Klausel-1.                % z.B. parse(G,W,B) :-...
...
% Erklärung, welche Prädikate/Köpfe ersetzt werden sollen:
eliminate( Kopf-1 ).      % z.B. (_ --> _)
...
% Definitionen der zu eliminierenden Prädikate/Köpfe:
def( (Kopf-1 :- Bedingung-1) ).  % z.B. (s --> np, vp).
...
```

Der partielle Auswerter `mix/2` nimmt eine Datei `<Quelle>.pl` dieses Formats und schreibt das Zielprogramm in `<Quelle>.pl.tr`.

Die partielle Auswertung

```
?- [mixer], pe(<Quellprogramm.pl>).
```

- überträgt *nicht* die Hilfsprädikate `eliminate/1` und `def/1`,
- überträgt *nicht* die durch `eliminate(Kopf)` oder `def((Kopf:-Rumpf))` als Hilfsregeln bekannten Regeln,
- optimiert die übrigen Regeln `(Kopf:-Rumpf)`, indem es darin die zu eliminierenden atomaren Bedingungen durch deren (optimierte) Definitionen ersetzt,
- und führt bei der Optimierung die durch `{{Goal}}` gekennzeichneten Teile aus, die also keine unbekannten Eingabeargumente haben dürfen.

Partiell auswerten

⟨*mixer.pl*⟩≡

```
:- module(mixer,[pe/1]).   % Moduldeklaration.

pe(File) :- % Filename including extension!
    consult(File),
    atom_concat(File,'.tr',Out),
    open(Out,write,Stream),
    absolute_file_name(File,AbsFile),
    ( predicate_property(mixer:Head,file(AbsFile)),
        \+ (Head = eliminate(_) ; Head = def(_)),
        \+ eliminate( Head ),
        clause(Head,Body),  % sucht Klausel zu Head
        mix((Head :- Body),Translation), % ersetzt
        portray_clause(Stream,Translation),
        fail
    ; close(Stream), write('Output in: '), write(Out) ).
```

mix(+Klausel,-Optimiert)

⟨*mixer.pl*⟩+≡

```
mix((Head :- Body), Expansion) :-
    !, mix(Body,NewBody),
    (NewBody = true -> Expansion = Head
     ; Expansion = (Head :- NewBody)).
mix((Clause1, Clause2), Expansion) :-
    !, mix(Clause1, ExpClause1),
    mix(Clause2, ExpClause2),
    simplify((ExpClause1, ExpClause2), Expansion).
mix((Clause1; Clause2), Expansion) :-
    !, mix(Clause1, ExpClause1),
    mix(Clause2, ExpClause2),
    simplify((ExpClause1; ExpClause2), Expansion).
mix({{Goal}},true) :-
    !, call(Goal). % Goal zur Compilezeit ausführen
```

⟨*mixer.pl*⟩+≡

```
mix(Literal, Expansion) :-
    eliminate(Literal),
    defined(Literal),          % Gibt es eine Definition,
    !, definition(Literal :- Body),    % so benutze sie
    mix(Body, Expansion).
mix(Literal, Expansion) :-
    Expansion = Literal,!.

defined(Literal) :-
    \+ \+ definition((Literal :- _Body)).
definition((Head :- Body)) :-
    ( def((Head :- Body)) ; def(Head), Body = true ).
```

Bem. `\+\+(Head)` beweist `Head` und hebt die Belegung wieder auf.

Vereinfache Konjunktionen und Disjunktionen

Vereinfachungen: `(A,true) ==> A` und `(A;fail) ==> A`
Aber nicht: `(A,fail) ==> fail` oder `(A;true) ==> true` !

⟨*mixer.pl*⟩+≡

```
simplify(((A,B),C), ABC) :- !,
    simplify((B,C), BC), simplify((A,BC), ABC).
simplify((true,A), A) :- !.
simplify((fail,_), fail) :- !.
simplify((A,true), A) :- !.
simplify((A,C), (A,C)) :- !.

simplify(((A;B);C), ABC) :- !,
    simplify((B;C), BC), simplify((A;BC), ABC).
simplify((fail;A), A) :- !.
simplify((A;fail), A) :- !.
simplify((A;C), (A;C)) :- !.
```

Beispiel 1: Beseitige ein Hilfsprädikat

⟨*quellprogramm1.pl*⟩≡

```
s(I,J) :- np(I,K), vp(K,J).
np(I,J) :- det(I,K), n(K,J).
vp(I,J) :- v(I,K), np(K,J).
vp(I,J) :- v(I,J).
vp(I,J) :- words([bringt],I,K), np(K,L),
           words([zum, 'Absturz'],L,J).
det(I,J) :- words([der],I,J).
det(I,J) :- words([das],I,J).
n(I,J) :- words(['Programmierer'],I,J).
n(I,J) :- words(['Programm'],I,J).
v(I,J) :- words([schreibt],I,J).

eliminate( words(_,_,_) ).
def(( words([],J,J) )).
def(( words([A|As],[A|K],J) :- words(As,K,J) )).
```

⟨*Definition von* `words` *einsetzen und beseitigen:*⟩≡

```
?- [mixer], pe('quellprogramm1.pl').
```

Das erzeugt folgende Datei, in der auch rekursive Aufrufe von `word/3` ersetzt sind:

⟨*quellprogramm1.pl.tr*⟩≡

```
n(['Programmierer'|A], A).
n(['Programm'|A], A).
v([schreibt|A], A).
vp(A, C) :- v(A, B), np(B, C).
vp(A, B) :- v(A, B).
vp([bringt|A], B) :- np(A, [zum, 'Absturz'|B]).
det([der|A], A).
det([das|A], A).
s(A, C) :- np(A, B), vp(B, C).
np(A, C) :- det(A, B), n(B, C).
```

Beispiel 2: Optimiere $p(t, Y, Z)$ zu $p_t(Y, Z)$

Mit dem Fall

```
mix({{Goals}},true) :- !,call(Goals).
```

kann man die Optimierung von Eigenschaften der Argumente abhängig machen (durch {{Tests(t)}}):

⟨*quellprogramm2.pl*⟩≡

```
q1(Y,Z) :- p(15,Y,Z).
q2(Y,Z) :- p(30,Y,Z).
q3(Y,Z) :- p(atom,Y,Z).

eliminate( p(X,_,_) ) :- number(X).
def( (p(10,Y,Y)) ).
def( (p(X,Y,Z) :-
        {{ X<10 }}, Z is min(X,Y)) ).
def( (p(X,Y,Z) :-
        {{ X>10, N is X-10 }}, p(N,Y,Z)) ).
```

Nur solche Aufrufe `p(X,Y,Z)` im Quellprogramms sollen ersetzt werden, bei denen `X` eine Zahl ist. Beim Ersetzen sollen die einzusetzenden Definitionen vom Wert des Arguments `X` abhängen, was durch teilweises Auswerten erreicht wird, z.B. in

```
def( (p(X,Y,Z) :-
        {{ X>10, N is X-10 }}, p(N,Y,Z)) ).
```

Da hier ein rekursiver Aufruf des zu eliminierenden Prädikats eingesetzt wird, wird auch der weiter optimiert.

Man erhält durch

```
?- [mixer], pe('quellprogramm2.pl').
```

die Zieldatei:

⟨*quellprogramm2.pl.tr*⟩≡

```
q1(Y, Z) :- Z is min(5, Y).
q2(Y, Y).
q3(Y, Z) :- p(atom, Y, Z).
```

Beispiel 3: Beseitige ein bekanntes Argument

Statt der Spezialisierung eines universellen Parsers,

`parse(+Grammatik,+Satz,-Baum)` $\mapsto$ `parseG(+Satz,-Baum)`

spezialisieren wir einfacher, ob `Atom` ein Blatt von `Term` ist:

`leaf(+Atom,+Term)` $\mapsto$ `leafTerm(+Atom)`

⟨*quellprogramm3.pl*⟩≡

```
leaf(Atom) :-
    ( leaf(Atom,f(a,g(b,c),d))
    ; tree(Term), leaf(Atom,g(e,Term)) ).

% leaf/2: normale Definition (bel.Term)
leaf(Atom,Term) :-
    leaves(Term,[],Leaves),
    memberchk(Atom,Leaves).
```

⟨*quellprogramm3.pl*⟩+≡

```
% leaf/2: bei Grundterm optimieren, durch
%         Auswerten von leaves(Term,[],Leaves)
eliminate( leaf(_,Term) ) :- ground(Term).
def(( leaf(Atom,Term) :-
        {{ leaves(Term,[],Leaves) }},
        memberchk(Atom,Leaves) )).

% leaves/2 nicht optimieren, nur benutzen:
eliminate( leaves(Term,_,_) ) :- ground(Term).
leaves([T|Ts],Acc,Leaves) :-
    !,leaves(T,Acc,TAcc), leaves(Ts,TAcc,Leaves).
leaves([],Acc,Acc) :- !.
leaves(Term,Acc,[Term|Acc]) :-
    atomic(Term), !.
leaves(Term,Acc,Leaves) :-
    compound(Term), !, Term =.. [_Fun|Args],
    leaves(Args,Acc,Leaves).
```

⟨*quellprogramm3.pl.tr*⟩≡

```
leaves([A|C], B, E) :- !,    % (Kopf :- Rumf) wird
        leaves(A, B, D),     % übernommen, wenn
        leaves(C, D, E).     % eliminate(Kopf), aber
leaves([], A, A) :- !.       % kein def( (Kopf :- ...) ).
leaves(A, B, [A|B]) :-       % vorhanden war.
        atomic(A), !.
leaves(A, C, D) :-
        compound(A), !,
        A=..[_|B],
        leaves(B, C, D).
leaf(B, A) :-
        leaves(A, [], C),
        memberchk(B, C).
leaf(A) :-                   % Optimierte Klausel:
        (   memberchk(A, [d, c, b, a])
        ;   tree(B),         % leaf/2 eliminiert
            leaf(A, g(e, B))% nicht eliminierbar
        ).
```