1 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

$$\varphi :- \varphi_1, \ldots, \varphi_n.$$

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

$$\texttt{fail} :- \texttt{true}.$$

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)

$$?- \ \varphi_1, \ldots, \varphi_n.$$

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 kann man zweierlei in Betracht ziehen:

(i) Die Antwort auf die Frage $?- \varphi$ ist der Wahrheitswert von φ.

(ii) Die Antwort auf die Frage $?- \varphi$ ist eine Begründung (bzw. Widerlegung) von φ.

PROLOG antwortet im Sinne von (ii).

Der PROPLOG-Interpreter

Um eine Anfrage $?-\psi.$ relativ zum Programm P zu beantworten, versucht PROLOG, die Aussage ψ mit Hilfe von P zu *beweisen*.

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.

Die Beantwortung erfolgt durch eine Funktion *provable*:

```
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
 end
```

Hilfsfunktionen:

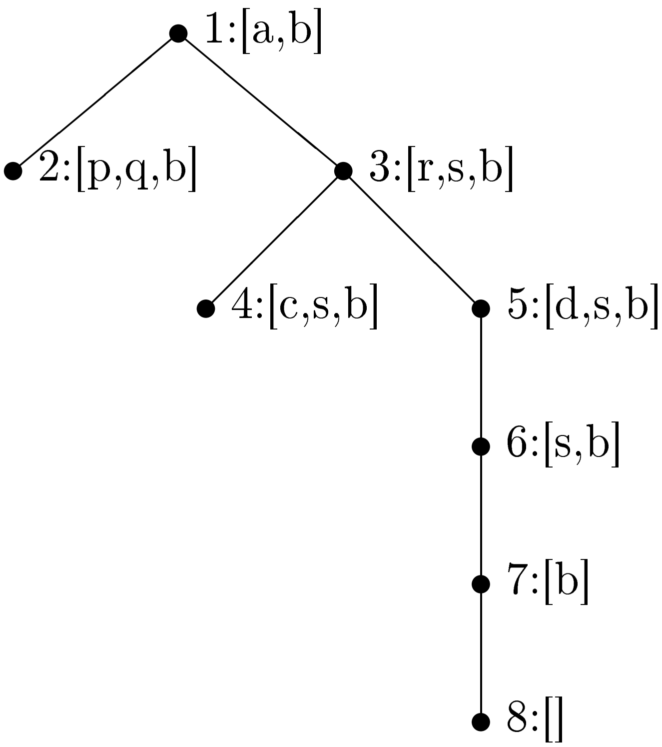
```
head(p :- q1,...,qn.) = p                head(p.) = p
body(p :- q1,...,qn.) = [q1,...,qn]      body(p.) = []

append([p1,...,pk],[q1,...qn]) = [p1,...,pk,q1,...,qn]
first([q1,...,qn]) = q1
rest([q1,...,qn]) = [q2,...,qn]
```

Beweissuche am Beispiel

Beispiel 1.1 Wir betrachten das 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.`

und stellen die Anfrage `?- a,b.` Der Suchbaum ist



Beim Ziel `1:[a,b]` paßt der Klauselkopf von $\varphi_1 =$ `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 $\varphi_2 =$ `a:- r,s.`

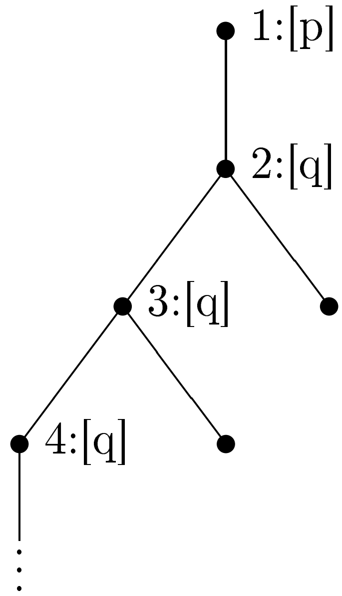
Scheiternde Beweissuche

Nicht auf jede Anfrage kann die Beweissuche *provable* eine Antwort finden:

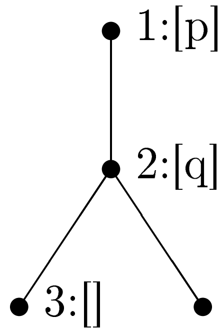
Beispiel 1.2 Betrachte die Anfrage `?- p.` an die Programme

```
p :- q.              p :- q.
q :- q.      und     q.
q.                   q :- q.
```

Der Suchbaum ist



bzw.



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:[]`.

Negation

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

Beispiel 1.3 Bei dem Programm

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

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

Die Antwort `No` auf die Frage $? - \varphi$ 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 $? - \ \varphi.$, 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 POPLOG (wie Prolog) um ein *not* erweitern durch:

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

Man nennt *not* die *negation as failure*.

Konjunktion und Disjunktion

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

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

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)` und `not(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)`.

Anwendungsbeispiel

Beispiel 1.4 *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.
```

Faktenlage 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.
```

2 DATALOG - Prolog mit atomaren Individuen

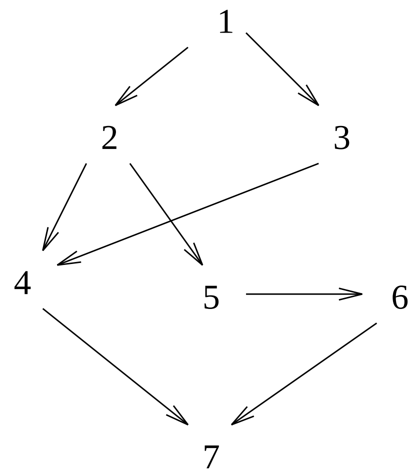
Syntax der Programme

Program	:=	Clauses
Clauses	:=	
	|	Clause Clauses
Clause	:=	Atom.
	|	Atom :- Atoms.
Atoms	:=	Atom
	|	Atom, Atoms
Atom	:=	RelSymbol
	|	RelSymbol(Terms)
Terms	:=	Term
	|	Term, Terms
Term	:=	VarSymbol
	|	ConstSymbol
RelSymbol	:=	[a-z]([a-zA-Z0-9_])*
ConstSymbol	:=	$[0\text{-}9]^+$
ConstSymbol	:=	[a-z]([a-zA-Z0-9_])*
VarSymbol	:=	[A-Z]([a-zA-Z0-9_])*
Goal	:=	?- Atoms.

Wege in kreisfreien Graphen

Beispiel 2.1 Gegeben sei ein endlicher, kreisfreier Graph $\mathcal{G} = (\{1, \ldots, m\}, kante)$, etwa

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



Betrachte folgende Definitionen von Wegen in einem Graphen:

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

Wir wollen an dieses Programm Fragen stellen wie

```
?- 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.

(i) Wähle aus G das nächste Teilziel, ψ.

(ii) 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 $S := \texttt{match}(\mathit{Kopf}(\varphi'_i), \psi)$, wenn überhaupt.

(iii) Durch (ii) wurde ψS mit

$$\mathit{Kopf}(\varphi'_i)S \ \texttt{:-}\ \mathit{Körper}(\varphi'_i)S$$

bewiesen, das erste Teilziel von $GS = [\psi S, \chi S, \ldots]$.

(iv) 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

(i) vergleiche φ mit ψ nach ihrer Form, und

(ii) belege dabei –bei gleichem Relationszeichen–

  (a) Variable in φ durch (spezialisierte) Teilterme der entsprechenden Stellen von ψ,
  (b) Variable in ψ durch (spezialisierte) Teilterme der entsprechenden Stellen von φ),

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

Interpretation von DATALOG

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 P ?– 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)

```
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
```

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

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

Hilfsbegriffe und -funktionen

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

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

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

$$\begin{array}{lcl} rS & := & r \\ r(s_1, \dots, s_n)S & := & r(s_1S, \dots, s_nS) \\ \\ cS & := & c \\ XS & := & \begin{cases} t, & \text{falls } t/X \in S, \\ X, & \text{sonst.} \end{cases} \end{array}$$

und die Anwendung von S auf eine Liste von Atomen durch

$$[\psi_1, \dots, \psi_n]S \quad := \quad [\psi_1 S, \dots, \psi_n S],$$

Beispiel 2.2

$$\begin{array}{ll} \multicolumn{2}{l}{[s(X,Y),\ r(X,c)]\{Y/X, d/Y\}} \\ = & [s(X,Y)\{Y/X, d/Y\},\ r(X,c)\{Y/X, d/Y\}] \\ = & [s(X\{Y/X, d/Y\}, Y\{Y/X, d/Y\}),\ \dots] \\ = & [s(Y,d),\ r(Y,c)]. \end{array}$$

Die Variablen werden simultan ersetzt, nicht nacheinander.
Auf einander folgende Substitutionen bilden eine neue:

$$\begin{array}{l} compose(\{t_1/X_1, ..., t_n/X_n\}, T) := \\ \quad \{t_1T/X_1, ..., t_nT/X_n\} \cup \{t/X \mid t/X \in T, X \notin \{X_1, \dots, X_n\}\} \end{array}$$

Match(ψ, ψ')

```
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
 end

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 Relationssymbol r haben, beginnt man mit der leeren Substitution

$$S := \{\}$$

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

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

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

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

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

$i = 3$: 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

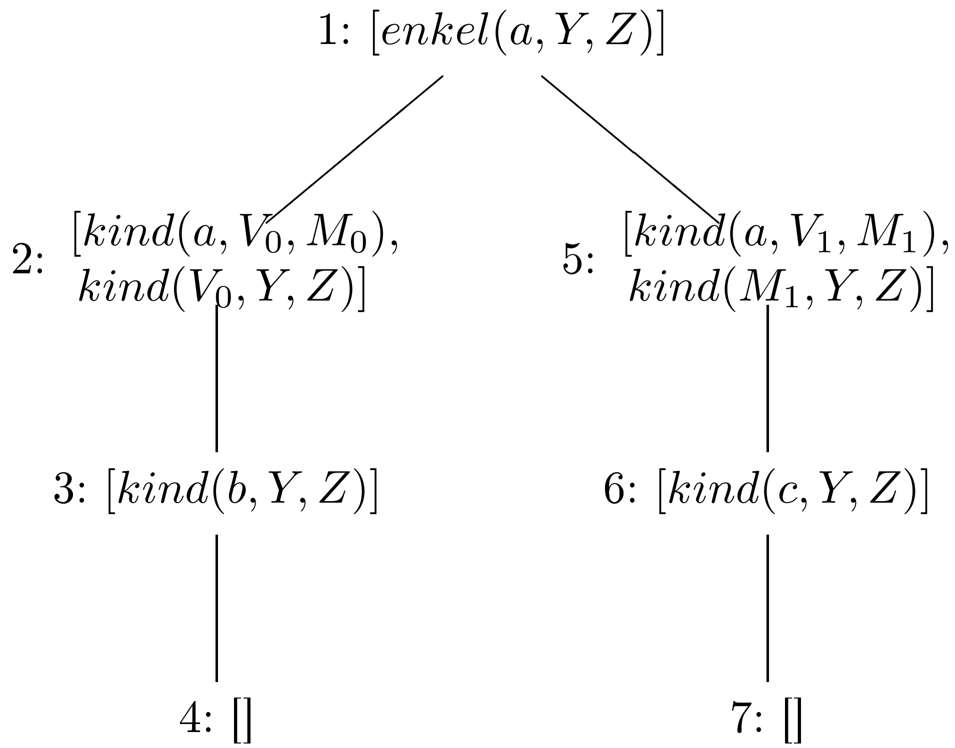
$$S := \mathit{No}.$$

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 (vgl. folgende Folie) zur Frage `?- enkel(a,Y,Z).` ist:



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

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

Der Suchbaum zu einer Frage

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

(i) Die Wurzel des Baums ist die Liste G von Zielen.

(ii) 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

$$(i_1)\ r(\vec{t_1}) :- \mathit{Rumpf}_1. \quad \dots \quad (i_m)\ r(\vec{t_m}) :- \mathit{Rumpf}_m.,$$

so verzweigt der Baum bei K zu m Nachfolgerknoten $K_1, \dots, K_m$ (den Wurzeln von m Teilbäumen).

An den Zweig von K zu K_j schreibt man die Nummer $j \in \{i_1, \dots, i_m\}$ 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]

(iii) Gibt es S_j, so schreibt man an den Knoten K_j die Ziele

$$[\mathit{Rumpf}_j S_j, G_2 S_j, \dots],$$

die aus denen bei K entstehen, indem man G_1 durch Rumpf_j ersetzt (bzw. G_1 wegläßt, wenn $\mathit{Rumpf}_j =$ `true` ist) und die Belegung S_j darauf anwendet.

(iv) Dann macht man dasselbe mit $K_1, \dots, K_m$ und ihren Zielen, bis man nur noch Knoten mit leeren Ziellisten bekommt. (Der Baum kann i.a. unendlich tief werden!)

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

Listen in Prolog

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

(i) der Konstanten `[]` für die *leere Liste*, und

(ii) 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:

```
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 von t um den Term s.

Vereinfachte Schreibweisen:

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

Beispiel 2.3 `[s(0)|[]]` und `[0|[s(0)|[]]]` sind Listen, in der Kurzschreibweise `[s(0)]` und `[0,s(0)]`.

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:

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

Umkehrung einer Liste:

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

Rekursive Implementierung:

```
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:

```
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 $\mathtt{rev}([\mathtt{X}|\mathtt{L}], \mathtt{Acc}, \mathtt{R})$

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

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

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

wobei

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

Die Bedingung $(*)$ sagt:

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

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. $g : \mathit{Term} \to \mathit{Term}$, so braucht man ein Maß $|\cdot|: \mathit{Term} \to \mathbb{N}$ für die Ein- und Ausgabewerte und nimmt statt (*)

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

Sortieren mit Quicksort

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

Idee:

(i) Wähle ein Element X aus L;

(ii) Zerlege L in eine Liste *Bigger* aller Elemente $Y \geq X$ und eine Liste *Smaller* aller Elemente $Y < X$;

(iii) Sortiere rekursiv die Listen *Bigger* und *Smaller*;

(iv) Bilde L' durch $append(Bigger', [X|Smaller'], L')$.

Programm:

```
/* quicksort(Liste,Sortiert)} */

quicksort([],[]).
quicksort([X|Xs],Ys) :-
        partition(X,Xs,Big,Small),
        quicksort(Big,Bs),
        quicksort(Small,Ss),
        append(Bs,[X|Ss],Ys).

/* partition(X,List,Big,Small) */

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

Quicksort mit Akkumulator

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

```
/* quicksort(Liste,Sortiert) */

quicksort(Xs,Ys) :-
        qusort(Xs,[],Ys).

/* qusort(Liste,Acc,Sortiert) */

qusort([],Acc,Acc).
qusort([X|Xs],Acc,Ys) :-
        partition(X,Xs,Big,Small),
        qusort(Big,Acc,Zs),
        qusort(Small,[X|Zs],Ys).

/* partition(X,List,Big,Small) */

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

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.

```
Yes
127 ?- trace.

Yes
127 ?- 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,[],[],[]) ?
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]
```

3 PURE PROLOG Prolog mit komplexen Individuen

Syntax der Programme

```
Program       :=  Clauses
Clauses       :=
               |  Clause Clauses

Clause        :=  Atom.
               |  Atom :- Atoms.

Atoms         :=  Atom
               |  Atom, Atoms

Atom          :=  RelSymbol
               |  RelSymbol(Terms)

Terms         :=  Term
               |  Term, Terms

Term          :=  VarSymbol
               |  ConstSymbol
               |  FunSymbol(Terms)

RelSymbol     :=  [a-z]([a-zA-Z0-9_])*
FunSymbol     :=  [a-z]([a-zA-Z0-9_])*
ConstSymbol   :=  [0-9]+
ConstSymbol   :=  [a-z]([a-zA-Z0-9_])*
VarSymbol     :=  [A-Z]([a-zA-Z0-9_])*

Goal          :=  ?- Atoms
```

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:

```
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:

```
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:

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

und damit das Maximum zweier natürlicher Zahlen durch:

```
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

relsymbol(r(t1,...,tn))  :=  r  =: relsymbol(r)
arg(r(t1,..,tn),i) := ti,    falls 0 < i < n+1
variable(X) := true,         variable(c) := false
```

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:

(i) vergleiche die Funktionszeichen von s und t, und –bei gleichem Funktionszeichen– vergleiche der Reihe nach die Argumentterme von s mit denen von t,

(ii) belege*) dabei überall

  (a) Variable in Teiltermen von s durch (spezialisierte) Teilterme der entsprechenden Stellen von t,

  (b) 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:

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

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:

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

$i = 2$: 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)

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

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

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

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

Sie wird wie folgt abgearbeitet:

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

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

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

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

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

$$6 : []$$

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_1S_2 ist

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

Also ist das Ergebnis ein Beweis von

$$\mathit{add}(s(s(0)), s(0), s(s(s(0)))),$$

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

Zur Terminologie

Für DATALOG wurde

match(s:Term, t:Term)

für den einfachen Fall von

unify(s:Term, t: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

(i) ein *Matcher* von r und t, falls $r = tS$ ist, und

(ii) 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:

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

4 PROLOG - Nichtlogische Konstrukte

Beschneiden des Suchbaums durch !

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

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

Bedeutung:

(i) 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;

(ii) 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.

(iii) Alternativen werden für die `q1,...,qk` gesucht (ggf. mit verschiedenen Belegungen der `Zs`), aber nicht für die `p1,...,pn`;

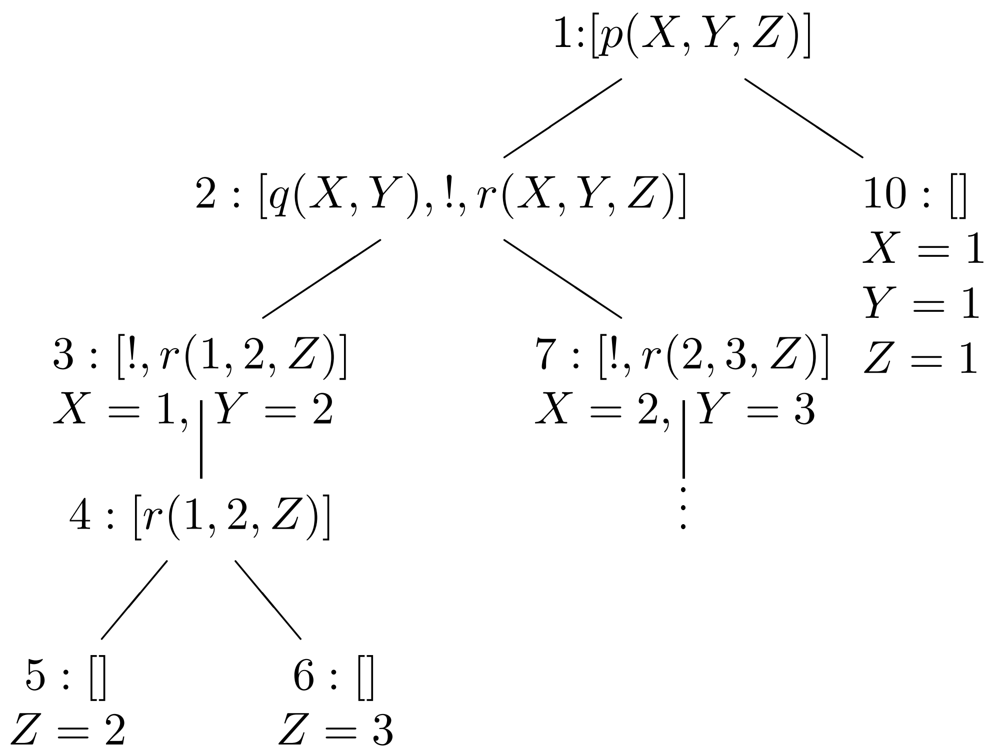
(iv) Es wird keine weitere Alternative für `praedikat(Us)`, z.B. über `r(Xs,Ys)`, probiert.

Beweissuche und !

Den Effekt eines ! zeigt folgendes 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:



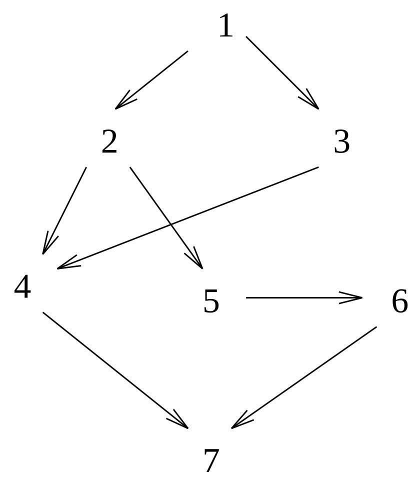
Durch Ausführen des ! beim Übergang zu Ziel 4 werden die Alternativen 7 und 10 abgeschnitten.

Die einzigen Lösungen sind (X,Y,Z) = (1,2,2) bzw. (1,2,3).

Beispiel für das Abschneiden der Suche

Beispiel 4.1 Sei $\mathcal{G} = (\{1, \ldots, m\}, kante)$ ein kreisfreier Graph, etwa wieder

```
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:

```
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:

```
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.:

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

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

```
?- 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:

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

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

```
?- 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.

```
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!

Beispiel 4.2 Wie oft kommt ein Term in einer Liste vor? (Eingabeargumente von `oftIn(+Term,+Liste,-Zahl)` sollten keine Variable enthalten.)

```
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:

```
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 scheinbare Alternativen, die nichts zur Lösung beitragen, weil sie scheitern oder eine Lösung wiederholen. Aber das kostet Zeit und Platz bei der Suche!

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

```
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

```
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) ?
   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:

```
    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) ?
    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:

```
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 ?
   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,

```
(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:

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

Beispiel 4.3 Fallunterscheidung bei `merge` mit `->` formuliert:

```
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) ?
   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:

```
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.

Eine andere Schreibweise für eine solche Schleife ist

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

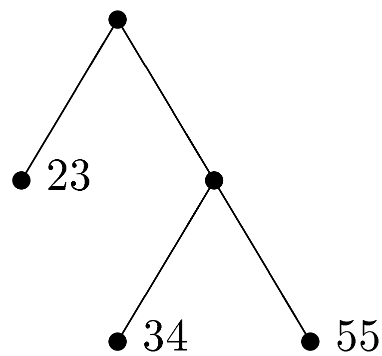
Beispiel 4.4 Anzeigen eines Graphen `kante/2`:

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

Datentypen in Prolog

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

Beispiel 4.5 Binäre Bäume mit Zahlen an den Blättern, etwa



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

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

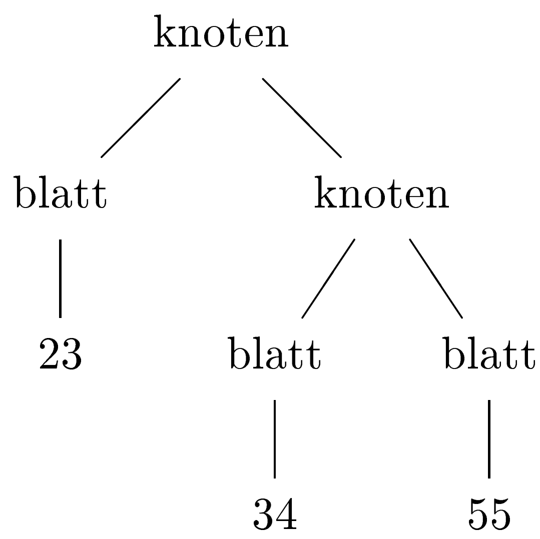
In Prolog haben wir dafür eine andere Notation:

```
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.

Datentypen (Forts.)

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



Wir unterscheiden also *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(blatt(N)) :- number(N).
baum(knoten(X,Y)) :- baum(X), baum(Y).
```

Der Beispielbaum ist also durch folgenden Term dargestellt:

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

Datentypen (Forts.2)

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

Praedikat `blaetter(+Baum,-Atomliste)`

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

```
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:

```
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 `Akk` kommen *zuerst* die Marken des *rechten* Teilbaums, damit die des linken vorne auf der Markenliste stehen!

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

5 STRÖME zur Ein- und Ausgabe

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

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

Erzeugen eines Stroms

Mit

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

erzeugt und *öffnet* man einen Strom zum Lesen oder Schreiben, 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 (Atom, 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.

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

```
close(Strom).
```

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.

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
...
write(Term),     % schreibt jetzt auf Datei
...
told.            % akt.Ausgabestrom wie vorher
```

Hier sollte `Datei` der Pfadname (als Prolog-Atom) einer existierenden Datei sein.

Dialog mit dem Benutzer

über die Ströme `user`, `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 den Benutzer am Bildschirm um eine Information fragen:

```
read(user,-Term)
```

Hier wird ein Prolog-Faktum eingelesen, d.h. die Antwort ist ein Term, gefolgt von einem Punkt.

Beispiel 5.1

```
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,'Danke, das war alles.').
```

Anwendung: Satzende-Erkenner und Tokenizer

- Lies Zeichenweise von einer Datei (oder der Konsole),
- Zerlege die Zeichenfolge in einzelne Wörter (Token),
- Wandle die Wörter in Prolog-Atome um,
- Erkenne das Satzende, und
- Gib den nächsten Satz als Atomfolge aus.

Einlesen von Sätzen aus einer Datei

```
% read_sentence(+Dateiname,-1.Satz als Atomliste)

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

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

Bis zum Satzende vom Strom lesen

```
% read_sentence(+Stream,-Sentence,+SeenChars)

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

% read_sentence(+Stream,-Sent.,+LastChar,+CharsRev)

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

% Sonderfälle: Pkt ist kein Satzende, also Weiterlesen
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,[Pkt|Seen]) :-
        abkuerzungsendeRev([Char,Pkt|Seen]),
        !, read_sentence(Stream,Sent,[Char,Pkt|Seen]).

% Fall Satzende erkannt: Char ignorieren, Satz ausgeben
read_sentence(Stream,Sentence,Char,[C|Seen]) :-
        satzende([C,Char]),
        !, reverse([C|Seen],Sentence).

% Sonst: Char sammeln und wieder zu read_sentence/3
read_sentence(Stream,Sentence,Char,Seen) :-
        read_sentence(Stream,Sentence,[Char|Seen]).
```

Tokenizer:
Zeichenreihe ↦ Atomfolge

Vorgehensweise:

(i) durchlaufe die Zeichenfolge von links nach rechts,

(ii) unterscheide dabei zwei Fälle:

  (a) das nächste Zeichen beginnt ein Wort (`nt`)
  (b) das nächste Zeichen setzt ein Wort fort (`at`)

(iii) stapele die schon erkannten Wörter (als Atome),

(iv) am Ende gib den umgekehrten Stapel aus.

stringToAtoms(+String,-Atomlist)

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

(a) tokenize(+String,nt,+TokensRev,-Atoms)

```
% Fall nt = suche Anfang den nächsten Tokens

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

Tokenizer
(bei Worttrennung, Zeilenende, Leerzeichen)

(b) tokenize(+String,at,+TokensRev,-Atoms)

```
% Fall at = das letzte Char lag in einem Token

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

tokenize([N|String],at,[[T|Cs]|Rs],Ts) :-
        worttrennung([T,N]),
        % Trennzeichen und Umbruch beseitigen:
        !, tokenize(String,at,[Cs|Rs],Ts).

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

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

Tokenizer
(in Abkürzungen und bei Sonderzeichen)

(b) tokenize(+String,at,+TokensRev,-Atoms)

```
% Sonderfaelle bei Abkuerzungen

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

tokenize([S|String],at,[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,nt,[Atom|Rs],Ts).

% Sonderfall bei Sonderzeichen

tokenize([S|String],at,[R|Rs],Ts) :-
        sonderzeichen(SAtom,S),
        !, tokenToAtom(R,Atom),
        tokenize(String,nt,[SAtom,Atom|Rs],Ts).
```

Tokenizer
(Token erweitern, Ende der Eingabe)

(b) tokenize(+String,at,+TokensRev,-Atoms)

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


tokenize([],at,[R|Rs],Ts) :-
        tokenToAtom(R,Atom),
        reverse([Atom|Rs],Ts).
tokenize([],nt,Rs,Ts) :- reverse(Rs,Ts).


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

In der Liste `[R|Rs]` ist das vordere Token `R` durch *die umgekehrte Reihenfolge seiner Buchstaben* dargestellt; daher muß `R` durch `tokenToAtom` erst umgekehrt werden.

Hilfsprädikate

Zeilenende, Worttrennung, Satzende

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

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

Punkte am Ende von Abkürzungen

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

Abkürzungen

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

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

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.").

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

suffixAbkuerzung("str.").
```

Sonderzeichen

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

% Ausser in Zahlausdruecken und Abkuerzungen
% sollten folgende Zeichen
% zu eigenstaendigen 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 Zitaten
sonderzeichen('(',40).
sonderzeichen(')',41).
```

Punkte am Ende von Ordnungszahlen

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

Beispiel

Prolog identifiziert einen String mit einer Liste von Character-Nummern:

```
?- "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:

```
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 ...

```
Call: (8) tokenize([D,r,'.',' ',L,e,e,?], nt, [], _G417) ?
Call: (9) leerZeichen(D) ?
Fail: (9) leerZeichen(D) ?
Call: (9) zeilenende(D) ?
Fail: (9) zeilenende(D) ?
Redo: (8) tokenize([D,r,'.',' ',L,e,e,?], nt, [], _G417) ?
Call: (9) sonderzeichen(_G414, D) ?
Fail: (9) sonderzeichen(_G414, D) ?
Redo: (8) tokenize([D,r,'.',' ',L,e,e,?], nt, [], _G417) ?
```

Der erste Buchstabe wird gelesen, und der zweite untersucht ...

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

Der zweite Buchstabe wird gelesen und der dritte untersucht ...

```
Call: (10) tokenize(['.',' ',L,e,e,?], at, [[r,D]], _G429) ?
Call: (11) worttrennung([r,'.']) ?
Fail: (11) worttrennung([r,'.']) ?
Redo: (10) tokenize(['.',' ',L,e,e,?], at, [[r,D]], _G429) ?
Call: (11) worttrennung([r,'.']) ?
Fail: (11) worttrennung([r,'.']) ?
Redo: (10) tokenize(['.',' ',L,e,e,?], at, [[r,D]], _G429) ?
Call: (11) zeilenende('.') ?
Fail: (11) zeilenende('.') ?
Redo: (10) tokenize(['.',' ',L,e,e,?], at, [[r,D]], _G429) ?
Call: (11) leerZeichen('.') ?
Fail: (11) leerZeichen('.') ?
Redo: (10) tokenize(['.',' ',L,e,e,?], at, [[r,D]], _G429) ?
Call: (11) punktZeichen('.') ? s
Exit: (11) punktZeichen('.') ?
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,?], at, [[r,D]], _G429) ?
Call: (11) punktZeichen('.') ? s
Exit: (11) punktZeichen('.') ?
```

Mit dem Punkt endet eine Abkürzung ...

```
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 als Token isoliert wird:

```
Call: (11) tokenize([' ',L,e,e,?], nt, ['Dr.'], _G459) ?

Call: (12) leerZeichen(' ') ?
Exit: (12) leerZeichen(' ') ?

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

Es beginnt ein neues Token ..

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

.. und wird fortgesetzt ..

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

.. durch einen dritten Buchstaben, aber der vierte Buchstabe ? ..

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

... beginnt ein neues Token ..

```
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') ?
```

.. so daß zwei weitere Token erkannt sind ..

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

... und das Ergebnis zurückgereicht wird:

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

   Exit: (7) stringToAtoms([D,r,'.',' ',L,e,e,?],
                               ['Dr.', 'Lee', ?]) ?
Yes
10 ?-
```

6 Horn-Klausel-Grammatiken (DCG)

Differenzlisten

Eine Teilfolge $K = [a_{i+1}, \ldots, a_j]$ 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.

Man kann $K = I - J$ als Umkehrung der Addition (= append)

$$K = I - J \iff K + J = I$$

von Listen verstehen.

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.

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

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

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

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

In der Darstellung mit Differenzlisten ist die Verkettung also

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

für *beliebige* Listen L, L', L''. Der Nutzen von Differenzlisten beruht darauf, daß man für die *spezielle* Wahl

$$\begin{aligned} &\underbrace{[a_{i+1}, \ldots, a_j|Bs]}_{As} - Bs \cap \underbrace{[b_{j+1}, \ldots, b_k|Cs]}_{Bs} - Cs \\ &= \quad \underbrace{[a_{i+1}, \ldots, a_j, b_{j+1}, \ldots, b_k|Cs]}_{As} - Cs \end{aligned}$$

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

Die Verkettung von Differenzlisten wird also durch

```
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.

Beipiel: Differenzlisten und Akkumulatoren

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

Mit Differenzlisten:

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

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

Mit Akkumulatoren:

```
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 in Prolog

- Kategorien als Eigenschaften von Wortfolgen
- Wörter als Einerlisten von Prolog-Atomen

Kontextfreie Grammatik

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

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

Direkte Übersetzung in Prolog

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

Yes
```

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 Regeln

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

```
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 hat man

```
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

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

s(I,J) :-          % von I bis J ein s
        np(I,K),
        vp(K,J).
```

Neue Übersetzung von Wörtern

Bei lexikalischen Regeln wie

```
det --> der.
```

erhält man aus

```
det([der]).
```

jetzt

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

beziehungsweise

```
det([der|J],J).
```

Ähnlich behandelt man Wörter in verzweigenden Regeln:

Ein Funktionsverbgefüge wie

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

wird übersetzt zu

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

Neu übersetzte Grammatik

```
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:

```
?- 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)

In Prolog geschieht eine solche Übersetzung implizit, wenn man das eingebaute `-->/2` benutzt. Als Grammatikregeln ist mehr erlaubt als bei kontextfreien Grammatiken:

Syntax der DCG-Regeln

DCG-Rule	:=	Category `-->` Constituents.
Category	:=	Atom
Constituents	:=	Category
	|	List
	|	`{` Term `}`
	|	(Constituents, Constituents)
	|	(Constituents; Constituents)
Atom	:=	RelSymbol
	|	RelSymbol(Terms)
List	:=	`[]`
	|	`[`Term|List`]`

Das soll heißen:

(i) Kategorien (= Nonterminale) können Terme, nicht bloß Konstante, sein: statt `np` etwa `np(Num,Kas)`,

(ii) Wortfolgen (= Terminale) sind durch Listen anzugeben,

(iii) mit { Code } kann Prolog-Code angegeben werden, der beim Parsen ausgeführt werden soll ,

(iv) Konstituenten können sequentiell (als nebeneinander stehend) oder alternativ verbunden werden.

Übersetzung einer DCG-Regel in eine Prolog-Klausel

Prolog übersetzt DCG-Regeln beim Einlesen in Klauseln, indem es an die Kategorien die Differenzlistenargumente ergänzt, z.B. wird *Cat* zu *Cat*(+Eingabeliste, −Restliste).

Syntaxregeln (Nebenbedingung: `r(X)`, Terminalsymbol: `[a]`)

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

werden übersetzt in

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

Lexikalische Regeln haben das Format

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

und werden übersetzt zu

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

Beispielgrammatik als DCG

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

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

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.

```
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

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

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

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

bzw.

```
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:

```
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:

```
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

Damit nicht Sätze analysierbar sind, sondern auch andere Ausdrücke, muß man bei diesen ggf. den Endpunkt in der Restliste lassen:

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

Erweiterung um syntaktische Merkmale

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

```
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:

```
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

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

oder in

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

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

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

und ergänzt die Grammatik um *eine* zusätzliche Regel:

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

oder gleich um die entsprechende Prolog-Klausel:

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

Oft verwendet man stattdessen Einträge der Form

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

und eine entsprechende Klausel pro Kategorie,

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