Universität Osnabrück
Fachbereich VII: Sprach- und Literaturwissenschaft
Computerlinguistik und künstliche Intelligenz
Sommersemester 98

Programmieren für CL & KI: Prolog

Leitung: Helmar Gust

Autor: Collin Rogowski

Inhaltsverzeichnis

1 Grundlagen

1.1 Fakten

Die einfachsten Ausdrücke in Prolog sind Fakten: `mann(hans).` `mann` wird Funktor genannt. Dieser Funktor hat einen Parameter $\Rightarrow$ `mann(hans)` ist eine einstellige Funktion.

1.2 Anfragen

Variablen in Prolog werden wie mathematische Variablen aufgefaßt und **nicht** wie in "herkömmlichen" Programmiersprachen (wie C oder Pascal) als Speicherzellen. Variablen unterscheiden sich von Fakten durch ihre Großschreibung. Beispiel für eine Anfrage:

```
mann(tobias).
mann(hans).
?- mann(X).
```

Diese Anfrage liefert `hans` und `tobias` als mögliche Lösungen.

1.3 Klauseln

Klauseln sind Fakten die nur unter einer bestimmten Bedingung wahr sind. Beispiel:

```
opa(hans, fritz) :-
  vater(hans, egon),
  vater(egon, fritz).
```

`:-` entspricht $\leftarrow$, der umgedrehten logischen Implikation. `,` entspricht $\wedge$, dem logischen und. Normalerweise formuliert man Klauseln allgemien, d.h. mit Variablen:

```
grossvater(X, Y) :-
  vater(X, Z),
  vater(Z, Y).
```

Diese Klausel entspricht folgendem logische Ausdruck:

$$\forall X, Y, Z : grossvater(X, Y) \leftarrow vater(X, Z) \wedge vater(Z, Y)$$

In Klauseln kann auch der Operator `;` benutzt werden. Er steht für $\vee$, das logische oder. Es gilt aber als schlechter Stil ihn zu verwenden. Klauseln mit `;` kann man immer umformulieren:

```
%schlecht:
vater(fritz, X) :-
  X = maria;
  X = franz.

%gut:
vater(fritz, maria).
vater(fritz, franz).
```

1.4 Strukturen

Strukturen erstellt mit Hilfe von Funkturen. Man braucht Strukturen nicht deklarieren, man schreibt sie einfach hin. Beispiel:

```
adresse(adr(fritz, mueller, hauptstrasse, 37, 49084,
            osnabrueck)).
```

`adresse` ist eine Funktion mit einem Parameter. `adr` ist ein Funktor. Die Anfrage `?- adresse(X)` liefert `X = adr(fritz, mueller, ...)`.

1.5 Listen

Listen basieren auf einer Cons-Zellen Struktur (wie in Lisp). Man erzeugt eine Liste, indem man einen beliebigen Funktor-Ausdruck schachtelt. Beispiel:

```
liste(.(a, .(b, .(c, .(d, [])))))).
% [] ist nil
```

Der Funktor `.` hat in Prolog eine besondere Bedeutung, er wird vom System als Funktor für Listen erkannt und deshalb wird die Ausgabe einer solchen Struktur geändert. Der obige Ausdruck wird wie folgt ausgegeben (und kann auch so eingegeben werden): `liste([a, b, c, d])`. Die allgemeinste Form einer Liste ist `[E | R]` wobei `X` das erste Element und `R` der Rest ist. `[E | R]` entspricht `.(E, R)`. Ein Algorithmus um zu überprüfen ob eine Element in einer Liste vorkommt sieht wie folgt aus:

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

2 Algorithmen in Prolog

2.1 umdrehen einer Liste

```
%langsame Version ohne Hilfsprädikat
reverse([], []).
reverse([X], [X]).
reverse([E | R], [L | R1]) :-
  reverse(R, [L, R2]),
  reverse(R2, R3),
  reverse([E | R3], R1).
```

2.2 mögliche Definition der natürlichen Zahlen

```
nat(1).
nat(suc(X)) :-
  nat(X).

+(1, X, suc(X)).
+(suc(Y), X, suc(Z)) :-
  +(X, Y, Z).

=(X, X).
=(+(X, Y), +(Y, X)).
```

2.3 das "generate - test" Verfahren

Man zählt zuerst einen (endlichen) Lösungsraum auf und verkleinert ihn dann immer weiter

```
individuum(i1).
individuum(i2).
individuum(i3).

.
.
.

taetigkeit(t1).
taetigkeit(t2).
taetigkeit(t3).

.
.
```

```
.

xtuty(X, Y) :-
  individuum(X),     %generate
  tateigkeit(Y),

  .
  .
  .
                     %test
```

3 natürliche Sprache $\rightarrow$ PL1 $\rightarrow$ Prolog

In Prolog gibt es die "closed world assumption":

- alles was nicht beweisbar ist, ist falsch
- alles was nicht im Programm steht existiert nicht

Faustregeln:

- alle Variablen die in Aussagen vorkommen sind All-Quantifiziert
- alle Variablen die in Anfragen vorkommen sind Existenz-Quantifiziert

3.1 Beispiel

Jeder Mann liebt eine Frau.

$$\forall x : mann(x) \rightarrow (\exists y : frau(y) \wedge lieben(x, y))$$
$$\exists y : frau(y) \wedge \forall x : (mann(x) \rightarrow lieben(x, y))$$
$$\exists y : \forall x : frau(y) \wedge (mann(x) \rightarrow lieben(x, y))$$

Der natürlichesprachige Satz ist ambique. Er kann bedeuten das alle Männer dieselbe Frau lieben, oder daß jeder Mann irgendeine Frau liebt.

```
lieben(X, Y) :-
  frau(Y),
  mann(X).
```

Bei der Übersetzung von PL1 nach Prolog hilft folgender Satz:

$$a \leftarrow \exists x : b \Leftrightarrow \forall x : a \leftarrow b$$

Den Allquantor braucht man auf Grund der "closed world assumption" nicht berücksichtigen. Den Existenzquantor kann man Skolemisieren (d.h. die Variable durch ein Individuum ersetzen). Nach der Skolemisierung entsteht folgendes Programm:

```
lieben(X, y) :-
  frau(y),
  mann(X).
frau(y).
```

4 Die Theorie hinter Prolog

Zu jeder Formel F der PL1 gibt es ein F' mit folgender Struktur:

$$F' = Q : ((a_{11} \vee a_{12} \vee \ldots \vee a_{1i}) \wedge (a_{21} \vee a_{22} \vee \ldots \vee a_{2i}) \wedge \ldots \wedge (a_{n1} \vee a_{n2} \vee \ldots \vee a_{ni}))$$

$$a_{ij} = p(t_1, \ldots, t_k) oder \neg p(t_1, \ldots, t_k)$$

$$Q = (\forall x_1 \ldots x_e \exists x \ldots \forall \ldots)$$

Folgende Umformungsregeln können angewandt werden:

$$a \rightarrow b \Rightarrow \neg a \vee b$$

$$\neg(a \quad op \quad b) \Rightarrow \neg a \quad op' \quad b$$

op	op'
$\wedge$	$\vee$
lor	$\wedge$

$$a \vee (b \wedge c) \Rightarrow (a \vee b) \wedge (a \vee c)$$

4.1 Erzeugung von Prolog-konformen Formeln

- elminiere $\rightarrow$ und $\leftrightarrow$
- $\neg$ nach innen holen
- $\vee$ nach innen holen
- $\wedge$ nach außen holen

4.2 Resolutionsverfahren

$$\{A\}, \{\neg A\} \rightarrow \{\}\{B, A\}; \{C, \neg A\} \rightarrow \{B, C\}$$

Das Verfahren sucht zwei Literale der Form:

$$\{B_1 \ldots B_n, A\}, \{C_1 \ldots C_n, \neg A\}$$

Hieraus wird ein neues Literal gebildet:

$$\{B_1 \ldots, B_n, C_1, \ldots C_n\}$$

Ensteht am Ende aller möglichen Anwendungen des Resolutionsverfahren die leere Menge, ist das System widerrsprüchlich.

$$\Sigma \models \phi \Leftrightarrow \Sigma \cup \{\neg\phi\}$$

Ist $\Sigma \cup \{\neg\phi\}$ widersprüchlich so gilt ϕ.

5 Algebra

Algebra in Prolog funktioniert mit Operator `is`. Dieser Operator berechnet den (möglichweise komplexen) Ausdruck auf seiner Rechten Seite, und unifiziert das Ergebnis mit seiner linken Seite. Beispiel:

```
Y = (3 + 5 * 7),   % ''syntaktischer Zucker'' für Y = +(3, *(5, 7))
X is Y.
```

Folgendes Konstrukt ist nicht möglich:

```
Y = +(Z, *(5, 7)), 38 is Y %geht nicht
```

Mit Hilfe der Algebra läßt sich sehr leicht ein Prädikat Länge formulieren:

```
laenge([], 0).
laenge([X, R], L) :-
  laenge(R, N),
  L is N + 1.
```

Dieses Prädikat funrkioniert nicht, da L beim Rekursionsabstieg wieder gelöscht wird. Hier die funktionierende (aber nicht so intuitive Lösung):

```
laenge([], N, N).
laenge([X | R], N, N2) :-
  N1 is N + 1,
  laenge(R, N1, N2).
```

Diese Lösung hat auch noch den Vorteil das sie Restrekursiv ist.

6 Satzgeneration und -analyse

6.1 erste Idee

```
s(Satz) :-
  np(Np),
  vp(Vp),
  append(Np, Vp, Satz).

.
.
.

n([frau]).
det([die]).

.
.
.

?- s([die, frau, studiert, computerlinguistik]).
?- s(X).
```

6.2 Verfahren mit Restlisten

Beispiel:

```
s([die, frau, studiert, computerlinguistik]) :-
  np([die, frau, studiert, computerlinguistik],
     [studiert, computerlinguistik]),
  vp([studiert, computerlinguistik], []).
```

Allgemeiner:

```
s(X, Y) :-
```

```
  np(X, Z),
  vp(Z, Y).

np(X, Y) :-
  det(X, Z),
  n(Z, Y).

n([frau | R), R).
```

6.3 Der DCG-Operator

Der Operator `-->` wird beim einlesen umgewandelt. Aus `ls --> rs1, ..., rsn` wird:

```
ls(X, Y) :-
  rs1(X, Z1),
  rs2(Z1, Z2),
  ...
  rsn(Zn, Y).
```

6.4 Beispiel für eine DCG in Prolog

```
s --> np, vp.
np --> det, n.
n --> [baum].
det --> [der].
vp --> iv.
iv --> [wächst].
```

7 Kontrollfluß

7.1 Der Cut

Wenn man versucht den Schnitt zweier Mengen zu berechnen kommt man leicht auf folgenden Algorithmus:

```
schnitt([], M, []).

schnitt([X | R], M, [X | R1]) :-
  member(X, M),
  schnitt(R, M, R1).
```

```
schnitt([X | R], M, R1) :-
  not member(X, M),
  schnitt(R, M, R1).
```

Diese Version liefert zwar das gewünschte Ergebnis ist aber uneffizient da zweimal `member/2` benutzt wird, obwohl einmal reichen würde. Wenn man das `not member(X, M)` einfach wegläßt produziert das System das gewünschte Ergebnis, verschwindet aber beim Backtracking im Nirvana. Man löst das Dilemma mit folgendem Code:

```
schnitt([], M, []).

schnitt([X | R], M, [X | R1]) :-
  member(X, M),
  !,
  schnitt(R, M, R1).

schnitt([X | R], M, R1) :-
  schnitt(R, M, R1).
```

Der Cut (`!`) sorgt dafür, daß wenn er bewiesen wurde keine weiteren Backtracking-Versuche mehr unternommen werden. Man unterscheidet zwischen roten und grünen Cuts. Rote Cuts ändern die Ausgabe des Programms, grüne Cuts ändern "nur" die Performance. Der oben eingefügte Cut ist also rot.
Der Code läßt sich jedoch durch einen grünen Cut noch weiter verbessern:

```
 schnitt([], M, []) :-
   !.

schnitt([X | R], M, [X | R1]) :-
  member(X, M),
  !,
  schnitt(R, M, R1).

schnitt([X | R], M, R1) :-
  schnitt(R, M, R1).
```

7.2 `not`

Das in Prolog eingebaute `not` ist kein logisches $\neg$. `not` ist meistens auf Prologebene definiert als:

```
not(A) :-
  A,
  !,
  fail.

not(A).
```

`fail` ist ein Prädikat was nie bewiesen werden kann, und damit Backtracking erzwingt. Die Variable an Goal Position ist eine sogenannte Meta-Variable. Das System versucht den Inhalt der Variablen zu beweisen.

7.3 Schleifen

7.3.1 rekursive Schleife

```
do_n(0, K) :-
  !.

do_n(N, K) :-
  N > 0,
  do_1(K),
  N1 is N - 1,
  !,
  do_n(N1, K).

do_1(K) :-
  K,
  !.

do_1(K).
```

7.3.2 Backtracking Schleife

```
loop(N) :-
  N > 0.

loop(N) :-
  N1 is N - 1,
  loop(N1).

.
.
.

loop(5).
```

```
.
.
.

fail.
```

8 Interne Struktur des Prolog-Systems

Man stellt sich ein Goal als eine Box mit zwei Ein- und zwei Ausgängen vor (Abbildung 1). Dies bezeichnet man als 4-Portmodell. Beispiel für die Darstellung

Abbildung 1: ein Goal

eines einfachen Prologprogramms durch das 4 Portmodell (Abbildung 2).

```
gv(X, Y) :-
  vater(X, Z),
  vater(Z, Y).

 gv(X, Y) :-
  vater(X, Z),
  mutter(Z, Y).
```

Ein Anfrage kann nun durch einen Baum dargestellt werden. Ausgehend von folgeden Fakten:

```
gv(hans, fritz).
vater(hans, egon).
vater(egon, fritz).
```

Ergibt die Anfrage `gv(hans, fritz)` die Baumstruktur in Abbildung 3.

9 eigener Prolog Interpreter

```
prove(true).

prove((G1, G2)) :-
```

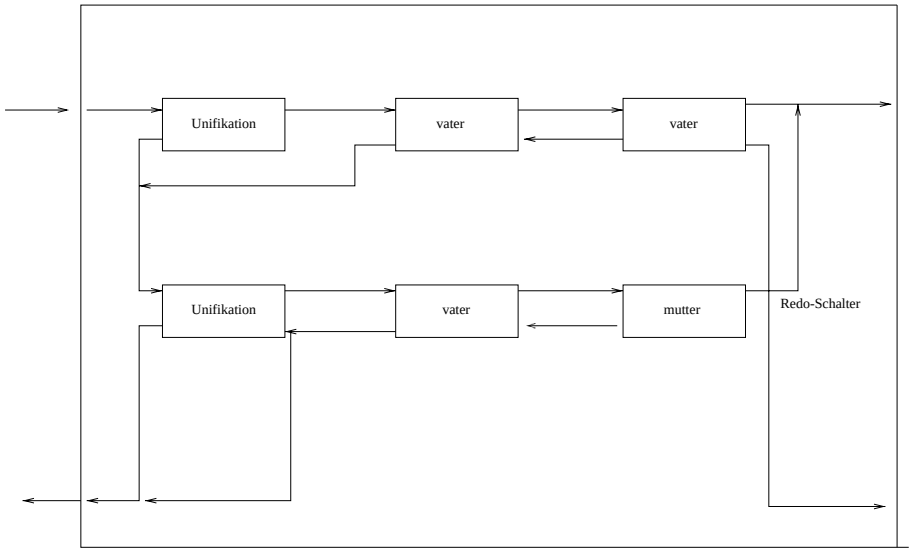

Abbildung 2: ein Goal mit sub-Goals

```
  !,
  prove(G1),
  prove(G2).

prove((G1 ; G2)) :-
  prove(G1).

prove((G1 ; G2)) :-
  !,
  prove(G2).

prove(G) :-
  clause(G, SG),
  prove(SG).
```

Dies ist ein eigener Beweiser. Von Prolog hat man die Unifikation und Verwaltung übernommen. Im folgenden Beispiel ist die Unifikation von Prolog erweitert:

```
prove(true).

prove((G1, G2)) :-
  !,
  prove(G1),
```

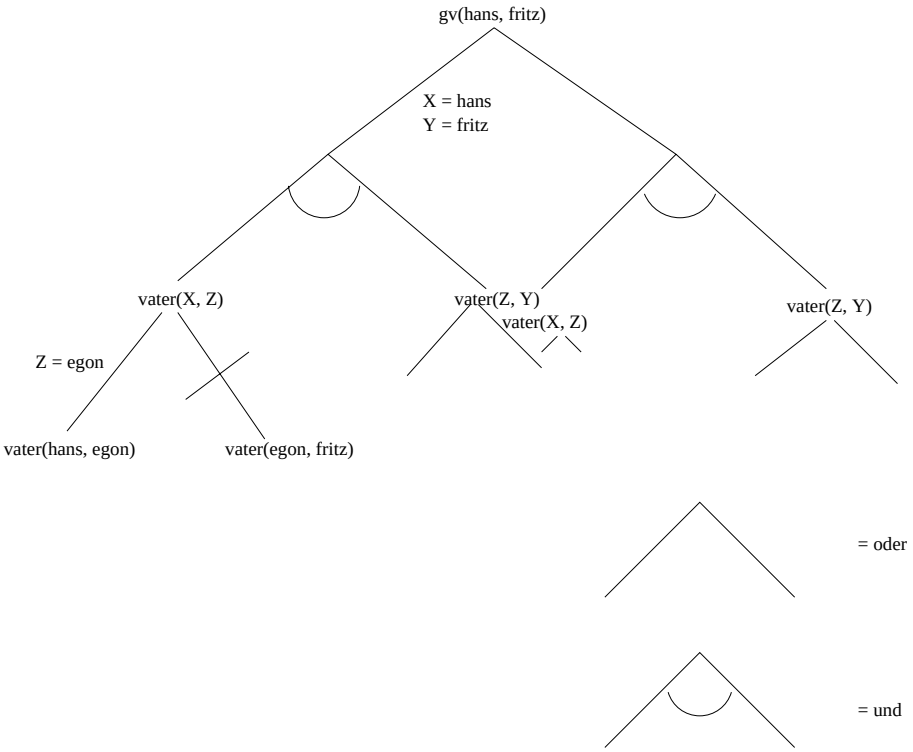

Abbildung 3: Baumdarstellung einer Anfrage

```
  prove(G2).

prove((G1 ; G2)) :-
  prove(G1).

prove((G1 ; G2)) :-
  !,
  prove(G2).

prove(G) :-
  unify(G, G1),
  clause(G1, SG),
  prove(SG).

unify(G1, G1) :-
  !.

unify(G1, G2) :-
```

```
  G1 =.. [F | AL1],
  G2 =.. [F | AL2],
  unify(AL1, AL2).

unify(G1, G2) :-
  G1 is G2.
```

10 Manipulation der Wissensbasis

Mit `assert` und `retract` kann man Fakten der Wissenbasis hinzufügen, oder wieder wegnehmen. Beispiel die Definition von `findall`:

```
findall(T, G, L) :-
  assert(fa([])),
  G,
  mretract(fa(L)),
  asserta(fa([T | L])),
  fail.

findall(T, G, L) :-
  mretract(fa(L)).

mretract(H) :-
  retract(H),
  !.
```