**Technische Universität München**
**Institut für Informatik**
**Prof. Tobias Nipkow, Ph.D.**
**Gerwin Klein**

**WS 2001/2002**
**Merkblatt 2**
**22. 10. 2001**

# Übungen zur Semantik von Programmiersprachen

# Kurze Einführung in Prolog

## Was ist Prolog?

Um Prolog zu verstehen, müssen ein paar Sätze zur Logikprogrammierung gesagt werden. Die Logikprogrammierung hat ihre Anfänge in den frühen siebziger Jahren. Sie wurde im wesentlichen von Kowalski [3] und Colmerauer eingeführt.

Die theoretischen Grundlagen der Logikprogrammierung gehen auf Arbeiten des Logikers Herbrand [2] zurück. Herbrand benutzte eine eingeschränkte Form der Prädikatenlogik erster Stufe, die sogenannte Horn-Logik, um Fragen der Berechenbarkeit von Funktionen zu studieren. Die Horn-Logik läßt nur spezielle prädikatenlogische Formeln zu, die sogenannten Hornklauseln (positiv definite Klauseln).

Die Horn-Logik wurde durch die Erfindung des Resolutionsprinzips durch Robinson [4] auch für den praktischen Einsatz interessant. Die fundamentale Idee von Kowalski und Colmerauer war, die Horn-Logik als Programmiersprache zu verwenden, wobei das Prinzip der Resolution den essentiellen Mechanismus der operationellen Semantik von Logikprogrammen darstellt.

Kurz gesagt ist der Grundgedanke von PROLOG: PROgramming in LOGic

Die klassische Einführung in die Programmierung in Prolog findet sich im Buch [1].

## Wie programmiert man in Prolog?

In Prolog formulieren wir einen Algorithmus, indem wir Objekte einführen und verschiedene Relationen beschreiben, die zwischen den Objekten bestehen.

Betrachten wir dazu die Aufgabe 1 auf dem Übungsblatt 1.

$$V = \{(s_0, s_1), (s_1, s_0), (s_1, s_2), (s_2, s_1), (s_2, s_3), (s_2, s_4), (s_3, s_2), (s_3, s_4), (s_4, s_2), (s_4, s_3)\}$$

Die zweistellige Relation $V$ wird durch die Aufzählung aller Paare beschrieben, die in Relation zueinander stehen. Wir legen also durch Aufzählung fest, welche Paare in $V$ sind. In Prolog spricht man hierbei von 'Fakten'. Eine Prolog Notation für obige Aufzählung wäre etwa:

## Fakten

```
v1rel(s0, s1).
v1rel(s1, s0).
v1rel(s1, s2).
v1rel(s2, s1).
v1rel(s2, s3).
v1rel(s2, s4).
v1rel(s3, s2).
v1rel(s3, s4).
v1rel(s4, s2).
v1rel(s4, s3).
```

Dadurch teilen wir Prolog mit, daß wir eine zweistellige Relation `v1rel` definieren. Für die Namensgebung ist unter anderem wichtig, daß Relationennamen klein geschrieben werden und alphanumerische Bezeichner verwendet werden. Die Syntax für Elemente ist etwas freizügiger, jedoch dürfen keine großen Anfangsbuchstaben verwendet werden.

Namen, die mit Großbuchstaben beginnen, sind Variablennamen! (dazu später mehr)

Durch die Zeile

```
v1rel(s0, s1).
```

teilen wir Prolog einen Fakt mit. Der Punkt nach `v1rel(s0, s1)` ist wichtig.

Mit obiger Aufzählung haben wir unser erstes Prologprogramm geschrieben. Nehmen wir an, daß die Aufzählung in der Datei `einfuehrung.pl` steht. Wir starten das Prologsystem mit dem Befehl

```
eclipse
```

und geben dann folgenden Befehl ein

```
[einfuehrung].
```

Das System liest dann die Datei `einfuehrung.pl` ein und erfährt so etwas über unsere Definition der Relation `v1rel`.

Nun können wir das System über die Relation `v1rel` befragen. Stellen wir die Frage

```
v1rel(s2, s3).
```

(der Punkt ist wichtig), antwortet das System mit

```
yes
```

Stellen wir hingegen die Frage

```
v1rel(s2, s2).
```

so antwortet das System mit

```
no (more) solution.
```

Eine solche Anfrage heißt im Prolog-Jargon Ziel (goal) oder Anfrage (question, query).

## Wie macht das Prologsystem das?

Prolog hat unsere Datei `einfuehrung.pl` gelesen und die Fakten in einer internen Datenbank abgespeichert. Wir stellen nun die Anfrage

```
v1rel(s2, s3).
```

am Systemprompt. Daraufhin geht Prolog Schritt für Schritt die Datenbank durch und vergleicht die vorhandene Fakten mit unserer Anfrage. Paßt ein Fakt, so gibt das System `yes` aus, andernfalls `no`.

## Variablen, Matching

Das ist bisher nicht sehr spannend! Nun wollen wir wissen, welche Städte `si` mit der Stadt `s3` in Relation `v1rel` stehen.

Wir stellen die Anfrage

```
v1rel(s3,X).
```

Prolog erkennt an der Schreibweise, daß `X` eine Variable sein soll.

Das System antwortet

```
X = s2      More? (;)
```

Wir tippen ';' ein, und das System antwortet

```
X = s4
yes.
```

Prolog ist wieder die einzelnen Fakten durchgegangen (in Aufschreibungsreihenfolge). Es teilt uns (nach Aufforderung mit ';') alle Belegungen der Variablen `X` mit, für die `v1rel(s3,X)` aus unserer Datenbank ableitbar ist. Dabei hat Prolog versucht, die Variable `X` in unserer Anfrage mit den Konstanten `si` aus unseren Fakten zur Deckung zu bringen. Man spricht von 'matching'.

## Regeln, Rekursion

Wenn wir unsere Relation `v1rel` genauer betrachten fällt uns auf, daß `v1rel` symmetrisch ist. Wenn also `v1rel(si,sj)` gilt, dann gilt auch `v1rel(sj,si)`.

Dies können wir auch Prolog miteilen. Wir schreiben unser Programm `einfuehrung.pl` um bzw. ändern es ab, indem wir folgende Fakten

```
v2rel(s0, s1).
v2rel(s1, s2).
v2rel(s2, s3).
v2rel(s2, s4).
v2rel(s3, s4).
```

und die 'Regel'

```
v2rel(Y,X):- v2rel(X,Y).
```

spezifizieren. Mit Fakten kennen wir uns bereits aus. Die Regel

```
v2rel(Y,X):- v2rel(X,Y).
```

ist folgendermaßen zu lesen:

Wenn wir wissen, daß `v2rel(X,Y)`, dann wissen wir auch daß `v2rel(Y,X)`. Oder anders ausgedrückt:

Um abzuleiten, daß `v2rel(Y,X)` gilt, genügt es zu zeigen, daß `v2rel(X,Y)` gilt.

Hier sehen wir das erste mal einen Hauch von Logik in unserem Prologprogramm aufscheinen. Weiterhin haben wir nun durch die Regeln eine Rekursion eigebaut. Um mit Hilfe

der Regel etwas über die Relation `v2rel` abzuleiten, benutzen wir schon Information über die Relation `v2rel`.

Stellen wir die Anfrage

```
v2rel(s4, s2).
```

so findet das System die Antwort auf unsere Anfrage nicht, indem es nur die ihm bekannten Fakten über die Relation `v2rel` prüft. Es muß auch die Symmetrie-Regel benutzen, die wir zusätzlich angegeben haben. Beide Versionen unserer Beschreibung der Relation $V$ sind gleichwertig.

Fakten und Regeln zusammen nennt man in der Logik (Horn-)Klauseln.

Wenn Prolog versucht, unsere Anfrage `v2rel(s4, s2)` mit Hilfe der Symmetrieregel zu beantworten führt es wieder Matching durch, diesmal aber in umgekehrter Richtung. Es versucht die Konstanten `s4` und `s2` in unserer Anfrage mit den Variablen `X` und `Y` in unserer Regel zur Deckung zu bringen. Genauer gesagt matcht es `s4` mit `Y` und `s2` mit `X`. Dies gelingt, und gemäß der Regel versucht es dann `v2rel(s2,s4)` abzuleiten. Dies gelingt mit Hilfe der Fakten.

## Unifikation

Nun kommt eine weitere Steigerung! Wir stellen die Anfrage

```
v2rel(s4, Z).
```

Auch hier muß die Symmetrieregel benutzt werden.

Das System antwortet:

```
Z = s2     More? (;)
```

Wir verlangen mit Eingabe `;` weitere Lösungen

```
Z = s3     More? (;)
Z = s2     More? (;)
Z = s3     More? (;)
Z = s2     More? (;)
Z = s3     More? (;)
Z = s2     More? (;)
...
```

Zwei Dinge sind zu erwähnen.

(a) Das System muß bei Anwendung der Symmetrieregel in beiden Richtungen matchen

`v2rel(s4, Z)` mit `v2rel(Y,X)`

Man spricht von Unifikation. Welche Belegung der Variablen `X`,`Y`,`Z` macht die beiden Ausdrücke gleich (unifiziert sie)?

Unifikationsprobleme und Algorithmen für ihre Lösung treten in vielen Bereichen der Informatik auf. Beispiele sind die automatische Typinferenz in funktionalen Sprachen oder das Finden von passenden Inferenzregeln in Theorembeweisern. Im Fall von Prolog ist das Unifikationsp roblem noch relativ einfach, und es gibt Unifikationsalgorithmen, die vollständig und korrekt sind (Stichwort: allgemeinster Unifikator).

(b) Das System generiert zyklisch die Antworten

```
Z = s2     More? (;)
Z = s3     More? (;)
```

Das liegt daran, daß beim Herleitungsversuch für `v2rel(s4, Z)` beliebig oft die Symmetrieregel verwendet werden kann. Die Rekursion terminiert nicht.

Prolog bietet die Möglichkeit, solche Merfachberechnung von Lösungen zu verbieten (Stichwort: cut `!`). Durch Verwendung des cut verläßt man aber die reine logische Programmierung. Wir gehen nicht näher darauf ein.

## Standard-Datentypen in Prolog; Listen

Bisher haben wir nur Konstanten `(s1,s2,...)` für Individuen und Variablen `(X,Y,Z..)` betrachtet. Nun beschäftigen wir uns mit dem Datentyp der Listen. Über diesem Datentyp lassen sich viele interessante rekursive Funktionen programmieren bzw. in Prolog induktiv Relationen spezifizieren.

Wichtig: In Prolog müssen alle Funktionen $f(x) = E(x)$ als Relationen `f-rel(x,E(x))` programmiert werden.

Notation für Listen:

`[]` bezeichnet die leere Liste `[x1,...,xn]` bezeichnet die endliche Liste der Elemente `x1` bis `xn`

`[c|XS]` bezeichnet die Liste, die mit dem Element `c` angeht und von einer Restliste `XS` gefolgt wird (Pattern in Klauselköpfen bzw. Konkatenation zur Bildung neuer Listen).

## Ein erstes Programm mit Listen; Test auf Vorkommen isin

Wir wollen ein Programm schreiben, das testet ob ein Element in einer Liste enthalten ist. In Prolog müssen wir unsere Aufgabe relational ausdrücken. Wir definieren die Relation `isin` zwischen Elementen und Listen, die ein Element `x` mit einer Liste `L` genau dann in Relation setzt, wenn `x` in `L` vorkommt. In Prolog geht das folgendermaßen:

```
isin(X,[X|YS]).
isin(X,[Y|YS]):- isin(X,YS).
```

Die erste Regel (Klausel) fragt ab, ob die Liste gerade mit dem Element beginnt, auf dessen Enthaltensein wir testen. Die zweite Regel prüft, ob das Element in der Restliste vorkommt (Rekursion).

Eine Anfrage

```
isin(x,[y,x,x]).
```

veranlaßt das System zur Antwort `yes.`

Die Anfrage

```
isin(x,[y,y,y]).
```

führt zur Antwort `no (more) solution.`

Wir können aber auch folgende Anfrage stellen:

```
isin(Z,[y,x,x]).
```

Für welche Belegung der Variablen `Z` ist die Relation `isin` erfüllt? Das System antwortet mit

```
Z = y     More? (;)

Z = x     More? (;)

Z = x     More? (;)

no (more) solution.
```

Hier sehen wir einen ganz neuen Aspekt der Kodierung von Funktionen als Relationen. Relationen sind nicht gerichtet, und somit können wir die Menge aller Urbilder einer Funktion erfragen. Diese Umkehrung von Funktionen ist nur möglich, wenn keine schmutzigen 'unlogischen' Konstrukte wie der cut in Programmen benutzt werden.

## Noch mehr Listenprogramme

```
% append von Listen

append([],L,L).
append([X|L1],L2,[X|L3]):-append(L1,L2,L3).

% Alle Vorkommen eines Elements aus einer Liste streichen

del(X,[],[]).
del(X,[X|XS],Z):- del(X,XS,Z).
del(X,[Y|XS],[Y|Z]):- X \= Y, del(X,XS,Z).

% Alle Duplikate aus einer Liste streichen
squeeze([],[]).
squeeze([X|XS],[X|Z]):- del(X,XS,Z1), squeeze(Z1,Z).

% direktes Bild einer Menge M unter Relation R
% Menge und Relation als Listen kodiert

% Bild eines einzelnen Elements
dom1(X,[],[]).
dom1(X,[(X,Y)|R],[Y|Z]):- dom1(X,R,Z).
dom1(X,[(W,Y)|R],Z):-   X \= W, dom1(X,R,Z).

% Wende dom1 auf ganze Menge an
dom2([],_,[]).
dom2([X|XS],R,Z):- dom1(X,R,B), dom2(XS,R,BS), append(B,BS,Z).

% mach Duplikate raus

dom(L,R,Z):- dom2(L,R,Z1),squeeze(Z1,Z).

% die Relation V als Liste
vrel([(s0, s1),(s1, s0),(s1, s2),(s2, s1),(s2, s3),(s2, s4),
      (s3, s2),(s3, s4),(s4, s2),(s4, s3)]).
```

Ein paar Anfragen:

```
- vrel(R), dom1(s0,R,Z).

R = [(s0, s1), (s1, s0), (s1, s2), (s2, s1), (s2, s3), (s2, s4), (s3, s2),
     (s3, s4), (s4, s2), (s4, s3)]
Z = [s1]     More? (;)


- vrel(R), dom1(s2,R,Z).
R = ...
Z = [s1, s3, s4]     More? (;)

- vrel(R), dom2([s1,s3],R,Z).
R = ...
Z = [s0, s2, s2, s4]     More? (;)

- vrel(R), dom([s1,s3],R,Z).
R = ...
Z = [s0, s2, s4]     More? (;)
```

# Literatur


[1] Clocksin, Mellish. Programming in Prolog, Springer 1984, 2nd Edition

[2] Herbrand. Researches in the Theory of Demonstrations. In *From Frege to Gödel: A Source Book in Matehmatical Logic, 1979-1931*, Harvard MA: Harvard University

[3] Kowalski. Predicate logic as a Predicate Logic, IFIP 1974

[4] Robinson. A Machine-Oriented Logic Based on the Resolution Principle, JACM, 12(1),23-41, 1965