# IV.2 Syntax von Prolog

Wednesday, 21 January, 2015 8:30

Prolog-Prog: Folge v. Klauseln

↑

Bestimmte log. Formeln

Klauseln müssen durch Zeilenumbruch oder Leerzeichen getrennt sein.

Klausel: Fakten + Regeln

Literal: Aussage, dass eine bestimmte Eigenschaft auf best. Objekte zutrifft.

Prädikat: Eigenschaft/Relation auf Objekten

Prädikate mit gleichem Namen u. unterschiedl. Stelligkeit gelten als verschieden.

weiblich/1

verheiratet/2

Prädikatssymbole sind Strings, die mit Kleinbuchstaben anfangen (dazu ex. vordef. Prädikate wie =, >, ...).

Literal: Prädikat wird auf Terme angewendet

Term: Variable oder Funktionsanwendg.

Variable: String, der mit Großbuchst. oder _ anfängt.

Spezielle Joker-Variable:

_

Falls sie mehrfach vorkommt, darf jedes Vorkommen unterschiedlich instantiiert.
Falls Anfrage _ enthält, wird keine Belegung dafür v. Prolog ausgegeben.

```
ist_Ehemann(X):-
    verheiratet(X,_).

mag(_,_).
    ↖ "Jeder mag jeden".

?- mV(renate,_).
true  ↑
    "Hat renate Kinder?"
```

Funktionssymbole:
haben eine Stelligkeit n. werden auf n Terme angewendet.
Bisher: Fkt-Symbole der Stelligkeit 0
monika/0 ↖ Konstanten
werner/0
Bsp: Prädikat geboren
1. Mögl geboren/4
geboren(monika, 25, 7, 1972).
Unschön.
2. Mögl geboren/2
Sollte Relation zwischen Person u. Datum sein
datum/3

datum(25, 7, 1972) ist
Fktsyms ein Term

geboren(monika, datum(25,7,72)).
Präd- ist ein Literal
symbol

?- geboren(X, _).
?- geboren(X, datum(_, 1, _)).

Fkt-Symbole = Strings
die mit Kleinbuchst.
anfangen (sowie
vordef. Symbole wie 1, 2, ...)

Überladen v. Fkt-Syms.
wie bei Präd-Symbolen
(datum/3 u. datum/2
sind verschieden).

Datenstrukturen in Prolog

Mit Funktionssymbolen
kann man eigene Daten-
strukturen konstruieren.

Datenstr. für IN:
zero/0
succ/1

zero ≙ 0
succ(zero) ≙ 1
succ(succ(zero)) ≙ 2
⋮

B: Wenn X+Y=Z,
dann X+s(Y) = s(Z).

Prolog-Programme definieren
Prädikate, die ausgewertet
werden können.
Funktionen werden in Prolog
nicht ausgewertet.
(Prolog-Funktionen ≙
Datenkonstruktoren in
Haskell. Werden nicht

ausgewertet, sondern dienen nur zur Konstruktion v. Daten objekten).

Lösung: add kann nicht als 2-stl. Fkt definiert werden (denn add(zero, s(zero)) würde nicht weiter ausgewertet werden).

Sondern: add muss als 3-stl. Prädikat definiert werden.

add(X, Y, Z) ⇐

"Die Addition von X und Y ist Z."

Generell: Zur Implem. einer n-stl. Fkt verwende (n+1)-stl. Prädikat.

?- a(s(Z), s(Z), U)

X = s(zero)
Y = zero
U = s(Z)

?- a(s(zero), zero, Z)

X' = s(zero)
Z = s(zero)

□ - Lösung:
U = s(s(zero))

Bei jeder Anwendg. einer Prog-Klausel werden die Var. der Prog-Klausel durch frische Var. ersetzt

Prog für Addition lässt sich auch zur Subtraktion verwenden

(gilt 1. u. 3. Arg vor u. sucht 2. Arg).

Multiplikation

$X \cdot (Y+1) = Z$, falls

$\underbrace{X \cdot Y}_{U} + X = Z$

Wie beeinflusst Reihenfolge der Literale das Erg. der Berechnung?

?- mult(s(z), s(z), U)

|

?- m(s(z), zero, U'), add(s(z), U', U)

| U' = zero

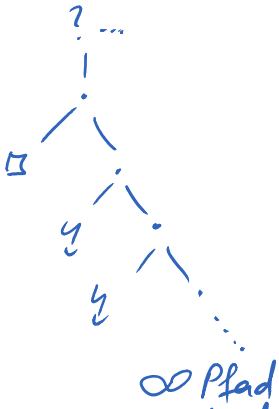
?- add(s(zero), zero, U)

| U = s(zero)

□

Erg: U = s(zero)

Hätte man die Literale in rek. mult-Regel im Rumpf vertauscht, dann hätte Beweisbaum die Gestalt:




1. Lösung wird gefunden, aber danach terminiert Prolog nicht, falls ; eingeg. wird.

Noch schlimmere Effekte entstehen, wenn rek.

Klausela vor nicht-
vek. Klausela kommen.

Listen

cons(1, cons(2, nil))

$\stackrel{?}{=}$

[1, 2]

Datenstruktur für
Listen ist in Prolog
vordef.

.(0, .(1, []))

Kurzschreibweise: =[0,1]

[$t_1, \ldots, t_n$ | $l$]

steht für

.($t_1$, .($t_2$, .... .($t_n$, $l$)..))

Außerdem:

[$t_1, \ldots, t_n$ | []] =

[$t_1, \ldots, t_n$]