

Term-Prädikate

Übersicht

Eingebaute Term-Prädikate

- ◆ Termsynthese und Termanalyse zur Laufzeit
 - ▶ `=./2, functor/3, arg/3`
 - ▶ `name/2` (siehe Folien zu Ein- und Ausgabe)
 - ▶ `atomic/1, atom/1, var/1, nonvar/1, compound/1, number/1, integer/1, float/1` (siehe Folien zur Syntax)
- ◆ Identität und Nicht-Identität von Termen
 - ▶ `==/2, \==/2`
- ◆ Standardordnung, Vergleichsprädikate und Sortierung von Termen
 - ▶ `@</2, @=</2, @>/2, @>=/2`
 - ▶ `sort/2`

Term-Prädikate - 1

Termsynthese und -analyse: `=./2`

- Das Prädikat `=./2 (univ)` verwandelt einen Term in eine Liste,

- ◆ deren erstes Element gleich dem Funktor des Terms ist,
- ◆ und deren restliche Elemente gleich den einzelnen Argumenten des Terms sind.

```
?- f(arg1,arg2) =.. Liste.  
Liste = [f,arg1,arg2]
```

- `=./2` kann auch aus einer Liste einen Term bauen:

```
?- Term =.. [f,arg1,arg2].  
Term = f(arg1,arg2)
```

Term-Prädikate - 2

Termsynthese und -analyse: `functor/3`

- Das Prädikat `functor(Term, F, S)` gelingt, falls gilt

- ◆ `Term` ist komplexer Term mit Funktor `F` und Stelligkeit `S`,

```
?- functor(hund(fido), Funktor, Stelligkeit).  
Funktor = hund, Stelligkeit = 1
```

- ◆ oder `Term` ist ein Atom/eine Zahl, wobei `Term = F` und `S = 0`

```
?- functor(a, F, S).  
F = a, S = 0
```

Mit `functor/3` lassen sich neue komplexe Terme bilden:

```
?- functor(Neu, b, 2).  
Neu = b(_22, _23)
```

Term-Prädikate - 3

Termsynthese und -analyse: `arg/3`

- `arg(N, Term, Arg)` gelingt, wenn `Arg` das `N`-te Argument von `Term` ist.

```
?- arg(3, f(a,b,c), Was).      ?- arg(1, f(a,b,c), a).  
Was = c                        yes  
yes
```

```
?- arg(2, f(a,b,c), d).  
no
```

- ▶ Weder `N` noch `Term` dürfen freie Variablen sein!

```
?- arg(N, f(a,d), d).  
{INSTANTIATION ERROR: arg(_36,f(a,d),_38) - arg 1}  
?- arg(2, Term, d).  
{INSTANTIATION ERROR: arg(2,_34,_35) - arg 2}
```

Term-Prädikate - 4

Identität und Nicht-Identität

- Das Prädikat `==/2` gelingt, wenn die zwei Argumentsterme identisch sind.

```
?- hund(f) == hund(f).  
yes
```

```
?- hund(Y) == hund(X).  
no
```

- Das Prädikat `\==/2` gelingt, wenn die zwei Argumentsterme nicht identisch sind. Variablen werden nicht gebunden!

```
?- hund(f) \== hund(X).  
yes
```

```
Term1 \== Term2 :-  
    \+ Term1 == Term2.
```

Mögliche Definition von `\==/2`

Standardordnung von Termen

- Das Prädikat `@</2` gelingt, wenn das erste Atom in alphabetischer Ordnung vor dem zweiten Atom steht.

```
?- adam @< eva.  
yes
```

```
?- adam @< adam.  
no
```

- Aber: Für *alle* Terme ist eine Standardordnung festgelegt

- Variablen (nach Alter) – *Nicht nach Variablennamen!*
- Fließkommazahlen (nach numerischem Wert)
- Ganzzahl (nach numerischem Wert)
- Atome (nach Zeichensatz-Kode)
- Komplexe Terme (nach Stelligkeit, Name des Funktors, Standardordnung der Argumente von links nach rechts)

```
?- C @< 1.2.  
yes
```

Weitere Ordnungsprädikate

Neben dem Term-Vergleichsprädikat `@</2` (*echt kleiner*) gibt es wie in der Arithmetik noch andere Vergleichsprädikate, die sich auf diese Standardordnung beziehen:

- `@=</2` (*gleich oder kleiner*)
- `@>=/2` (*größer oder gleich*)
- `@>/2` (*echt größer*)

Achtung: Es gibt keine Prädikate `@<=/2` und `@>=/2`...

Standardsortierung von Termen

- Das eingebaute Prädikat `sort/2` bringt eine beliebige Termliste in die Standardordnung:

```
?- sort([fo(0,2),X,fo,A,-9,-1.0,1,fi,fi(1,1,1),X=Y],L).  
L = [X,A,-1.0,-9,1,fi,fo,X=Y,fo(0,2),fi(1,1,1)]
```

- Identische Terme dürfen im 2. Argument nur einmal erscheinen.

► Die Liste im 2. Argument ist also eine geordnete Menge!

```
?- sort([1,'1',1],L).  
L = [1,'1']  
yes
```

```
?- sort([1,1],[1,1]).  
no
```