Terme

Heute:

- Terme vergleichen
- Struktur von Termen
- Operatoren

Termgleichheit: $==$

?- a $==$ a.

yes

?- a $==$ b.

no

?- X $==$ Y.

no

?- X $==$ X.

yes

?- X $==$ a.

no

Termungleichheit: \==

```
?- a \== a.
   no
?- a \== b.
   yes
?- X \== Y.
   yes
?- X \== X.
   no
?- X \== a.
   yes
```

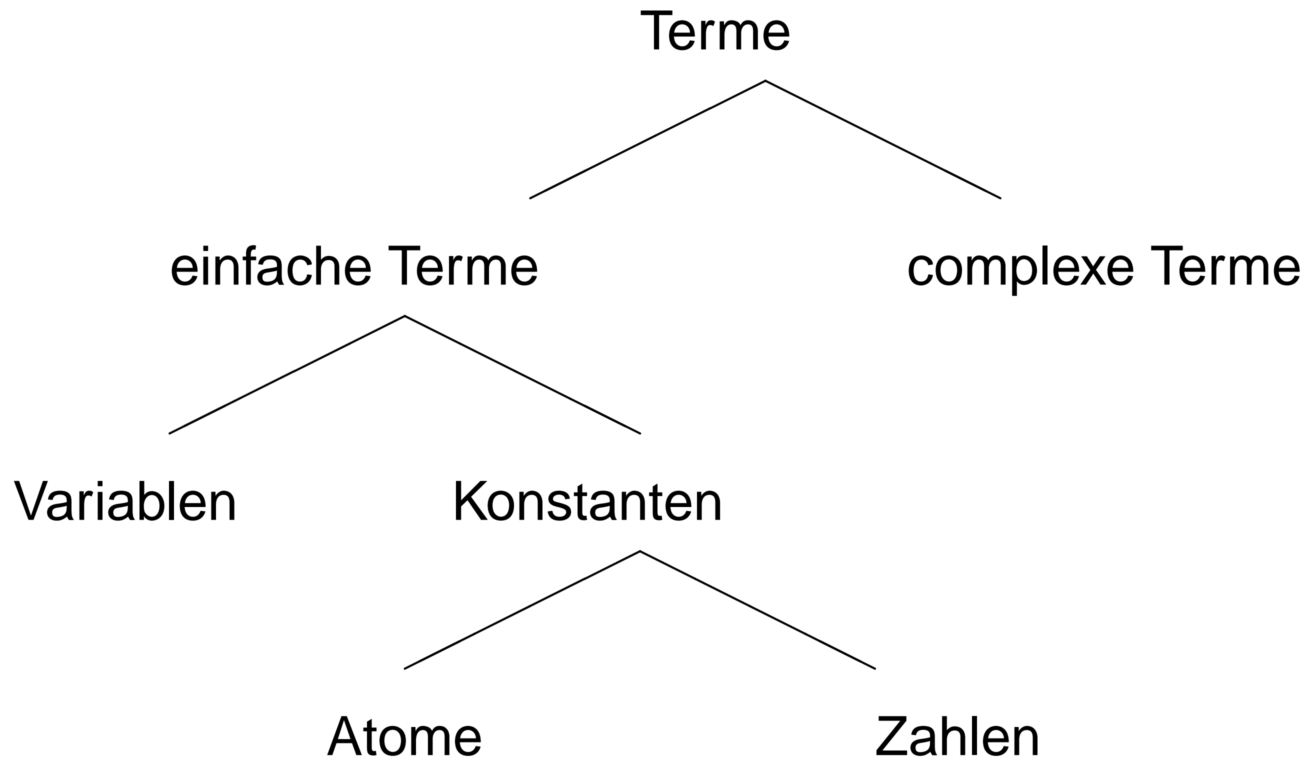
== vs. =

```
?- a == X.
no

?- a = X.
yes

?- X == Y.
no

?- X = Y.
yes

?- X=Y, X==Y.
yes
```

Vergleichsoperatoren

Operator	Negation	Vergleichtyp
=	\=	Unifikation
=:=	=\=	Aritmetische Gleichkeit
==	\==	Termgleichkeit

Terme in Prolog



Terme analysieren

Prädikat	testet ob sein Argument
atom/1	ein Atom ist
integer/1	eine natürliche Zahl ist
number/1	eine Zahl ist
atomic/1	eine Konstante ist
var/1	uninstanziiert ist
nonvar/1	instanziiert ist

Beispiele

```
?- atom(a).
yes

?- atom(7).
no

?- atom(X).
no

?- atomic(play(mia,piano)).
no

?- atomic(7).
yes
```

```
?-var(X).
yes

?- var(x).
no

?- nonvar(X).
no.

?- X=g, nonvar(X).
yes
```

Die Struktur der Terme

Bei der Verarbeitung von komplexen Termen braucht man Prädikate, die:

- den Zugriff auf den Funktor erlauben
- die Stelligkeit eines komplexen Terms zurückgeben
- den Zugriff auf ein bestimmtes Argument eines komplexen Terms erlauben
- den Zugriff auf alle Argumente eines komplexen Terms erlauben

Das Prädikat functor/3

Der Zugriff auf Funktor und Stelligkeit eines komplexen Terms ist ermöglicht von dem Prädikat **functor/3**:

```
?- functor(f(a,b),F,A).
F=f
A=2
?- functor(a,F,A).
F=a
A=0
?- functor([1,2,3],F,A).
F='.'
A=2
```

functor/3

Mit dem Prädikat **functor/3** ist es auch möglich komplexe Terme zu konstruieren:

```
?- functor(T,f,5).
T=f(_1,_2,_3,_4,_5).
yes
```

Aber:

```
?- functor(C,f,A).
ERROR: Arguments are not sufficiently instantiated
?- functor(C,F,3).
ERROR: Arguments are not sufficiently instantiated
```

Das Predikat arg/3

Das Predikat **arg/3** ermöglicht den Zugriff auf die Argumente eines komplexen Terms:

```
?- arg(1, play(mia, guitar), Result).
Result = mia
?- arg(2, play(mia,piano), Result).
Result = piano
?- arg(2, play(mia), Result).
no
```

Anwendung von arg/3

Mit dem Prädikat **arg/3** es ist möglich Elemente in komplexe Termen zu instanziieren:

?- arg(1, loves(X,mia), vincent).

X = vincent.

?- arg(1, loves(X,Y), vincent), arg(2, loves(X,Y), mia).

X=vincent

Y=mia

Das Prädikat ’=..’/2

Das Prädikat **’=..’/2** (univ) ermöglicht die Umwandlung von einem komplexen Term in eine Liste:

```
?- f(a,b,c,d) =.. X.
X= [f,a,b,c,d]

?- X =.. [f,a,b,c,d]
X = f(a,b,c,d)

?- play(mia,X) =.. Y
X=_23
Y=[play, mia, _23]
```

Anwendung von '=..'

Wir wollen ein Prädikat **term_a_to_b/2** definieren, das in allen Argumenten eines komplexen Terms das Atom *a* durch das Atom *b* ersetzt:

z.B. f(a, b, a, c) ⟶ f(b, b, b, c))

```
/*
Das Prädikat term_a_to_b/2 wandelt alle a Atome
eines Terms in b Atome um
*/
term_a_to_b(Term_a,Term_b):-
                        Term_a =.. [H|Ra],
                        list_a_to_b(Ra,Rb),
                        Term_b =.. [H|Rb].
```

/*

Das Prädikat **list_a_to_b/2** wandelt alle a Atome einer Liste in b Atome um

*/

```
% Basisklausel
list_a_to_b([],[]).

% Rekursive Klausel
list_a_to_b([H|T],[b|R]) :- H==a, list_a_to_b(T,R).

list_a_to_b([H|T],[H|R]) :- H\==a, list_a_to_b(T,R).
```

Aufgaben

1. Definiere ein Prädikat **islist/1**, das testet, ob sein Argument eine Liste ist.
2. Definiere ein Prädikat **isComplexTerm/1**, das testet, ob sein Argument ein komplexer Term ist.
3. Definiere ein Prädikat **check_term/1**, das testet , ob alle Argumente eines komplexen Terms Atome sind.

Benutzerfreundliche Notation von Operatoren

Interne Darstellung	Benuzterfreundliche Darstellung
+(3,2)	3+2
is(X,+(2,3))	X is 2+3
+(3,-(2))	3 + -2
$>$(4,3)	4$>$3

Eigenschaften von Operatoren

Operatoren werden durch die folgende Eigenschaften definiert:

- Typ
- Priorität
- Assoziativität

Typ

Der Typ eines Operators bestimmt die Beziehung zwischen dem Operator und seinen Argumenten; d.h. ob der Operator zwischen, vor oder nach seinen Argumenten geschrieben werden muss.

Es gibt drei Typen von Operatoren:

- infix Operatoren $\longrightarrow$ xOy

  z.B. Operator +: 3+4

- prefix Operatoren $\longrightarrow$ Ox

  z.B. Operator -: -2

- postfix Operatoren $\longrightarrow$ xO

Priorität

Die Priorität eines Operators bestimmt die Beziehung des Operators zu anderen Operatoren. Durch die Priorität wird der Hauptoperator eines Ausdrucks bestimmt.

Gegeben die folgenden Operatoren, geordnet nach aufsteigender Priorität:

Prec(O1) $>$ Prec(O2) $>$ Prec(O3)

Dann wird der Ausdruck:

x O2 y O3 z O1 w

wie folgt geklammert:

O1(O2(x,O3(y,z)),w)

Priorität (forts.)

Zum Beispiel:

Prec(is) $>$ Prec(+) $>$ Prec(*)

X is 3 + 2*4 $\Rightarrow$ is(X,+(3,*(2,4)))

Priorität von Termen

- Atome, Variablen und Zahlen haben die Priorität 0.
- Die Priorität von komplexen Termen wird durch die Priorität des Hauptoperators bestimmt.

Assoziativität

Die Assoziativität bestimmt die Klammerung der Argumente in einem Ausdruck, in dem mehrere Operatoren mit der gleichen Priorität vorkommen.

Operatoren können in Bezug auf ihre Assoziativität:

- rechtsassoziativ sein
- linksassoziativ sein
- nicht assoziativ

Rechtsassoziative Operatoren

Das ein Operator rechtsassoziativ ist bedeutet, dass die Priorität des linken Arguments kleiner als die des Operators sein muss, so dass der Ausdruck von rechts angefangen geklammert wird:

x O_1 y O_1 z O_1 w $\longrightarrow$ O_1(x,O_1(y,O_1(z,w)))

3 + 4 + 5 + 6 $\longrightarrow$ (3 + (4 + (5 + 6)))

Linksassoziative Operatoren

Das ein Operator linksassoziativ ist bedeutet, dass die Priorität des rechten Arguments kleiner als die des Operators sein muss, so dass der Ausdruck von links angefangen geklammert wird:

x O_1 y O_1 z O_1 w $\longrightarrow$ O_1(O_1(O_1(x,y),z),w)

3 + 4 + 5 + 6 $\longrightarrow$ (((3 + 4) + 5) + 6)

Nicht assoziative Operatoren

Operatoren können auch nicht assoziativ sein, d.h. beide Argumente des Operators müssen eine kleinere Priorität haben als der Operator. In diesem Fall kann der Ausduck nicht geklammert werden:

```
?- 2 =:= 3 == =:=(2,3).
ERROR: Syntax error: Operator priority clash
```

```
?- (2 =:= 3) == =:=(2,3).
yes
```

Operatoren selber definieren

Prolog erlaubt dem Programmierer Operatoren selber zu definieren.

:- op(Priorität, Typ&Assoziativität, Name).

Priorität $\in \{$1,...,1200$\}$

Typ&Assoziativität $\in$:

$\{$xfx, xfy, yfx$\}$ wenn f infix ist

$\{$fx,fy$\}$ wenn f prefix ist

$\{$xf,yf$\}$ wenn f postfix ist

x bedeutet: Die Priorität dieses Arguments ist kleiner als die Priorität des Operators.

y bedeutet: Die Priorität dieses Arguments ist kleiner oder gleich der Priorität des Operators.

Beispiele

```
:-op(500, yfx, +).
:-op(700, xfx, =).
:-op(700, xfx, is).
```

- X is 3 + 2 $\Rightarrow$ (X is (3+2)) $\Rightarrow$ is(X, +(3,2))
- 3 + 2 + 4 $\Rightarrow$ ((3+2)+4)

Beispiel

Wir wollen die Syntax von Prolog mit einem neuen Postfixoperator erweitern, der testet ob sein Argument eine Liste ist oder nicht.

Wir werden in zwei Schritten vorgehen:

- Operator deklarieren,
- Bedeutung zuweisen

```
% Syntaktische Erweiterung
:-op(500, xf, is_a_list).

% Bedeutungszuweisung:
is_a_list([]).
is_a_list(L) :- functor(L, '.', 2), arg(2, L, T), is_a_list(T).
```

```
?- [a,b,c] is_a_list.

yes

?- [] is_a_list.

yes

?- a(b,c) is_a_list.

no

?- is_a_list([1,2,3]).

yes
```

Zusammenfassung

Heute haben wir gesehen

- was für eingebaute Prädikate es zur Termverarbeitung gibt,
- was Operatoren sind und
- wie man Operatoren selber definieren kann.

Nächste Woche Montag (01.12.) Cut und Negation

Übungsaufgaben: Die Übungen sind auf der Webseite.