Prolog

9. Kapitel: Terme

Dozentin: Wiebke Petersen

Kursgrundlage: Learn Prolog Now (Blackburn, Bos, Striegnitz)

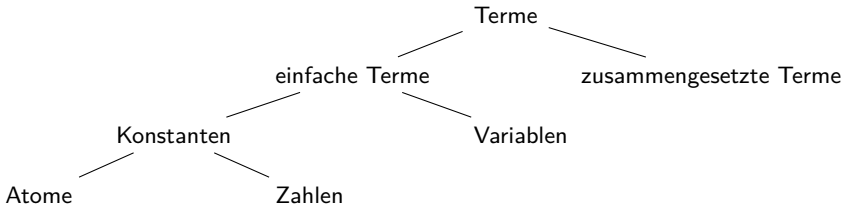
Zusammenfassung Kapitel 8

- Wir haben Parameter kennengelernt und diese eingesetzt,
  - um grammatische Constraints wie z.B. Kongruenz zu erfassen,
  - um mithilfe eines Zählers die kontextsensitive Sprache $a^n b^n c^n$ zu modellieren.
- Wir haben gesehen, wie wir DCGs mit Extrazielen anreichern können. Dies ist möglich, da DCGs nur *notational sugar* sind.
- Mit Parametern und Extrazielen ist es möglich die Grenzen von kontextfreien Grammatiken zu verlassen.

- **Keywords**: Parameter, Extraziele
- **Wichtig**: Durch Parameter und Extraziele sind DCGs mächtiger als kontextfreie Grammatiken.
- **Ausblick Kapitel 9**: Terme und Operatoren

Wiederholung: Terme

- Die grundlegende Datenstruktur in Prolog sind **Terme** (*terms*).
- Sie sind entweder **einfach** oder **zusammengesetzt**.
- Einfachen Terme in Prolog sind **Konstanten** (*constants*) und **Variablen** (*variables*)
- Die Konstanten sind **Atome** (*atoms*) und **Zahlen** (*numbers*).
- Zusammengesetzte Terme werden auch **komplexe Terme** oder **Strukturen** genannt.



Wiederholung: Zusammengesetzte bzw. komplexe Terme

- Zusammengesetzte bzw. komplexe Terme bestehen aus einem **Funktor** (*functor*) und beliebig vielen **Argumenten** (*arguments*).
- Der Funktor ist immer ein Atom.
- Die Argumente sind einfache oder komplexe Terme.
- Bsp. komplexer Term: `liebt(popeye,spinat)`
- Bsp. komplexer verschachtelter Term:
  `befreundet(X,vater(vater(popeye)))`
- Unter der **Stelligkeit** (*arity*) eines komplexen Terms versteht man die Anzahl seiner Argumente.

Wiederholung: besondere Terme – Listen und arithmetische Ausdrücke

Listen sind komplexe Terme mit Funktor `'[|]'`:

```
?- [a,b]='[|]'(a,'[|]'(b,[])).
true.
```

arithmetische Ausdrücke sind ebenfalls komplexe Terme:

```
?- 3+4 = +(3,4).
true.
?- 5*(3+4) = *(5,+(3,4)).
true.
?- (X is 3+4) = is(X,+(3,4)).
true.
?- (3<4) = <(3,4).
true.
```


Wiederholung: Matching-/ Unifikationsoperator

Der Matchingoperator „=“ gelingt, wenn die Argumente unifiziert werden können.

Der negierte Matchingoperator „\=“ gelingt genau dann, wenn „=“ nicht gelingt.

```
?- a = a.
true.
?- [a,food(eis)] = [a,food(X)].
X = eis.
?- 3+5 = 3+X.
X=5.
?- 3+5 = 5+X.
false.
```

```
?- a \= a.
false.
?- [a,food(eis)] \= [a,food(X)].
false.
?- 3+5 \= 3+X.
false.
?- 3+5 \= 5+X.
true.
```


Wiederholung: arithmetischer Gleichheitsoperator

Der arithmetische Gleichheitsoperator „=:=“ erzwingt die arithmetische Auswertung beider Argumente und prüft sie anschließend auf Gleichheit.

Der arithmetische Ungleichoperator „=\=“ gelingt genau dann, wenn die Ergebnisse ungleich sind.

```
?- a =:= a.
domain error
?- 3+5 =:= 5+3.
true.
?- 3+5 =:= 3+X.
instantiation error
?- 3+5 =:= 8.
true.
```

```
?- 3+5 =\= 8.
false.
?- 3+5 =\= 3*4.
true.
?- 3+5 =\= 3+X.
instantiation error
```


Vergleich von Termen

- Der Gleichheitsoperator für Terme „==“ vergleicht zwei Terme auf Gleichheit.

```
?- a == a.
true.
?- X == a.
false.
?- X = a, X==a.
true.
?- 2+3 == +(2,3).
true.
?- (4>5) == >(4,5).
true.
?- [a|[b]] == '[|]'(a,'[|]'(b,[])).
true.
?- 2+3 == 3+2.
false.
```

- Der Ungleichheitsoperator für Terme „\==“ gelingt genau dann, wenn „==“ nicht gelingt.

```
?- a \== a.
false.
?- a \== X.
true.
```

Übersicht Matching- und Vergleichsoperatoren

Operator	Negation	Vergleichstyp
=	\=	Unifikation
=:=	=\=	arithmetische Gleichheit
==	\==	Termgleichheit

▸ Übung1 ▸ Übung2

Analyse von nicht zusammengesetzten Termen

Mit den folgenden eingebauten Prädikaten kann man den Typ eines nicht zusammengesetzten Terms überprüfen:

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

```
?- atom(a).
true.
?- number(7.3).
true.
?- var(X).
true.
```

```
?- integer(7).
true.
?- atomic(7).
true.
?-nonvar(a).
true.
```

▸ Übung

Analyse zusammengesetzter Terme

- Die Struktur eines zusammengesetzten Terms besteht aus (1) dem Funktor, (2) der Stelligkeit und (3) dem Typ der Argumente.
- Die folgenden eingebauten Prädikate ermöglichen die Analyse der Struktur zusammengesetzter Terme:
  - Das Prädikat `functor/3` ermöglicht den Zugriff auf den Funktor und die Stelligkeit eines komplexen Terms.
  - Das Prädikat `arg/3` ermöglicht den Zugriff auf einzelne Argumente eines komplexen Terms.
  - Zusätzlich kann man mit dem **univ** genannten Prädikat „`=../2`“ einen komplexen Term in eine Liste umwandeln.

Das Prädikat: functor/3

Das Prädikat `functor/3` ermöglicht den Zugriff auf den Funktor und die Stelligkeit eines komplexen Terms.

```
% functor(+ComplexTerm, ?Functor, ?Arity)
% functor(?ComplexTerm, +Functor, +Arity)
?- 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
```

Das Prädikat: functor/3

Prolog wäre nicht Prolog, wenn man das Prädikat `functor/3` nicht auch zur Generierung komplexer Terme einsetzen könnte.

```
?- functor(T,f,4).
T=f(_A,_B,_C,_D).
```

Allerdings muss entweder das erste oder das zweite und dritte Argument instantiiert sein:

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

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

▸ Übung

Testen ob ein Term zusammengesetzt ist

Wie können wir testen, ob ein Term zusammengesetzt ist?

```
complexterm(X):-
    nonvar(X), % Variablen sind nicht zusammengesetzt
    functor(X,_,A),
    A > 0. % die Stelligkeit muss groesser 0 sein.
```

```
?- complexterm(X).
false.
?- complexterm(4).
false.
?- complexterm(mag(popeye,food(X))).
true.
```


Das Prädikat: arg/3

Das Prädikat `arg/3` ermögliche den Zugriff auf einzelne Argumente eines komplexen Terms.

```
% arg(+Number, +ComplexTerm, ?NthArgument)
?- arg(1, mag(popeye, spinat), Argument).
Argument = popeye.
?- arg(2, mag(popeye,spinat), Argument).
Argument = spinat.
?- arg(2, essen(spinat), Argument).
false. % scheitert, da essen/1 nur ein Argument hat.
```

Das Prädikat `arg/3` kann auch zur Instantiierung von Argumenten genutzt werden.

```
?- arg(1, liebt(X,olivia), popeye).
X = popeye.
```

▸ Übung

Das univ-Prädikat: =../2

- Das univ genannte Prädikat `=../2` ermöglicht die Umwandlung eines komplexen Terms in eine Liste und umgekehrt.
- Der Funktor des komplexen Terms wird zum ersten Element der Liste.
- Das univ-Prädikat kann auch als Infixoperator verwendet werden.

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

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

?- spielt(olivia,X) =.. Y, X= 20.
X = 20.
Y = [spielt, olivia, 20].

?- 6-8+9 =.. X.
X = [+,6-8,9].
```

▸ Übung

Bildschirmausgabe: write_canonical/1 und write/1

Das Prädikat write_canonical/1 gibt die Struktur eines (zusammengesetzten Terms) auf dem Bildschirm aus:

```
?- write_canonical(5+6*3).
+(5,*(6,3))
true.
?- write_canonical(5-3 < 4+7).
<(-(5,3),+(4,7))
true.
```

Das Prädikat write/1 schreibt einen Term in der externen Notation auf den Bildschirm:

```
?- write(5+6*3).
5+6*3
true.
?- write(5-3 < 4+7).
5-3 < 4+7
true.
```


strukturierte Bildschirmausgabe: nl/0 und tab/1

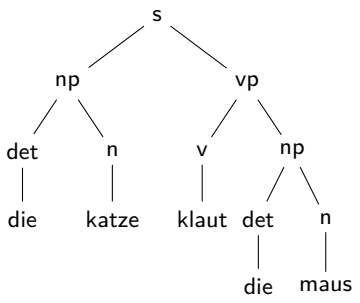
Das Prädikat nl/0 erzeugt einen Zeilenumbruch und das Prädikat tab/1 erzeugt die angegebene Menge an Leerzeichen auf dem Bildschirm.

```
?- write(a),write(b),write(c),write(d).
abcd
?- write(a),nl,write(b),tab(2),write(c),tab(5),write(d).
a
b  c     d
```


Ausgabe von Bäumen: pprint/1

In Kapitel 8 haben wir gesehen, wie man einen Ableitungsbaum als komplexen Prologterm repräsentieren kann:
s(np(det(die), n(katze)), vp(v(klaut), np(det(die), n(maus))))

Der dazugehörende Baumgraph ist:



Ziel: gut lesbare Darstellung erzeugt von Prolog:

```
s
   np
      det
         die
      n
         katze
   vp
      v
         klaut
      np
         det
            die
         n
            maus
```


Ausgabe von Bäumen: pprint/1

Das Prädikat pprint/1 nimmt als Argument einen Baum in Prolog-Term-Notation und erzeugt dazu eine Ausgabe auf dem Bildschirm, die besser lesbar ist.

```
% Initialisierung des Tab-Zaehlers.
pprint(Term):- pprint(Term,0).

% Baum drucken.
pprint(Term,N):-
   Term =.. [F|Args],      % Struktur zu Liste.
   tab(N), write(F), nl, % Ausgabe des Mutterknotens.
   N1 is N+3,              % Tabulator erhoehen.
   pprintl(Args,N1).       % Unterbaeume drucken.

% Unterbaeume drucken.
pprintl([H|T],N):-
   pprint(H,N),            % Drucke eine Schwester.
   pprintl(T,N).           % Drucke die anderen Schwestern.
pprintl([],_).             % Termination.
```

Operatoren – externe und interne Notation

Operatoren sind Prädikate, die eine zusätzliche nutzerfreundliche externe Notation erlauben:

interne Notation	nutzerfreundliche externe Notation
`+(1,2)`	`1+2`
`is(X,+(2,3))`	`X is 2+3`
`+(8,-(2))`	`8 + -2`
`>(4,1)`	`4 > 1`
`==(a,a)`	`a == a`
`=(X,a)`	`X = a`

Operatoren werden durch den **Typ**, die **Priorität** und die **Assoziativität** definiert.

Typen von Operatoren

Der Typ eines Operators bestimmt ob der Operator vor, zwischen oder nach seinen Argumenten geschrieben wird.

- **Infix-Operatoren** wie `=`, `<`, `is`, `+`, `\==` usw. sind zweistellig und werden zwischen die Argumente geschrieben (`3<4`).
  x Op y
- **Präfix-Operatoren** wie `-`, `+` usw. sind einstellig und werden vor das Argument geschrieben (z.B. `-3`).
  Op x
- **Postfix-Operatoren** sind einstellig und werden hinter das Argument geschrieben.
  x Op

Präzedenz von Operatoren

- Die Präzedenz eines Operators legt fest, in welcher Reihenfolge die Operatoren binden.
- Der Operator mit der höchsten Präzedenz ist der **Hauptoperator** eines Ausdrucks. Beispiel:
  - Operatoren geordnet nach absteigender Präzedenz:
    $prec(op1) > prec(op2) > prec(op3)$

```
?- write_canonical(x op2 y op3 z op1 w).
op1(op2(x,op3(y,z)),w)
true.
```

- **Hinweis:** einfache Terme und Terme in Klammern haben die Präzedenz 0. Die Präzedenz von komplexen Termen wird durch die Präzedenz des Hauptoperators bestimmt.

Assoziativität von Operatoren

Die Assoziativität bestimmt die Klammerung der Argumente in einem Ausdruck mit mehreren Operatoren gleicher Präzedenz.

- **links-assoziative** Operatoren fordern, dass ihr linkes Argument eine kleinere Präzedenz hat (d.h. Prolog klammert den Ausdruck von links):

```
?- write_canonical(x op1 y op1 z op1 w).
op1(op1(op1(x,y),z),w)
true.
```

- **rechts-assoziative** Operatoren fordern, dass ihr rechtes Argument eine kleinere Präzedenz hat (d.h. Prolog klammert den Ausdruck von rechts):

```
?- write_canonical(x op1 y op1 z op1 w).
op1(x,op1(y,op1(z,w)))
true.
```

- **nicht-assoziative** Operatoren fordern, dass beide Argumente eine kleinere Präzedenz haben (Prolog kann solche Ausdrücke nicht klammern):

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

Definition eigener Operatoren

Eigene Operatoren können definiert werden:

```
:-op(Praezedenz, Typ (+Assoz.), Name)
```

- `Praezedenz` $\in \{1, ...2000\}$
- `Typ (+Assoz)` $\in \begin{cases} \{xfx, xfy, yfx\} \text{ wenn f Infix ist} \\ \{fx, fy\} \text{ wenn f Präfix ist} \\ \{xf, yf\} \text{ wenn f Postfix ist} \end{cases}$
  - x bedeutet das die Präzedenz dieses Arguments kleiner als die des Operators ist.
  - y bedeutet das die Präzedenz dieses Arguments kleiner oder gleich der des Operators ist.
- `Name`: Name des Operators oder Liste von Operatornamen, die alle dieselbe Eigenschaft bekommen sollen.

Definition bestehender Operatoren

```
:- op( 1200, xfx, [ :-, --> ]).
:- op( 1200, fx, [ ?- ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1000, xfy, [ ',' ]).
:- op( 700, xfx, [ =, is, =.., ==, \==, =:=, =\=, <, >, =<, >= ]).
:- op( 500, yfx, [ +, -]).
:- op( 500, fx, [ +, - ]).
:- op( 400, yfx, [*,/]).
:- op( 300, xfx, [ mod ]).
```

$3+4+5$:

+(3, +(4 ,5))					+(+(3, 4), 5))				
		500	0	0		500	0	0	
500	0	500			500	500	0		

$3+4*5$:

*(+(3, 4), 5))					+(3, *(4 ,5))				
	500	0	0				400	0	0
400	500			0	500	0	400		

▸ Übung

Beispiel

4	−	5	/	6	+	7	/	8	<	9	mod	10
	yfx		yfx		yfx		yfx		xfx		xfx	
0	500	0	400	0	500	0	400	0	700	0	300	0

<(	+(	−(	4,	/(	5,	6)),	/(	7,	8)),	mod (	9,	10))
				yfx			yfx			xfx		
				400(	0,	0)	400(	0,	0)	300(	0,	0)
		yfx										
		500(	0,	400		)						
	yfx											
	500(	500,					400		)			
xfx												
700(	500,									300		)

Definition eines eigenen Operators

Definition eines neuen Infixoperators `in`, welcher testet ob etwas Element einer Liste ist (analog zum `member`/2-Prädikat).

```
:-op(500,xfx,in).

in(X,[X|_]).
in(X,[H|T]):-
    in(X,T).
```

Wir können nun Anfragen wie diese stellen:

```
?- 5 in [3,7,w,5,l].
true.
?- k in [3,7,w,5,l].
false.
```

Zusammenfassung Kapitel 9

- Wir haben verschiedene Prädikate zur Analyse von zusammengesetzten Termen kennengelernt:
  - `functor`/3
  - `arg`/3
  - `=../2` (univ)
- Wir haben gesehen, wie wir verschiedene Ausgaben auf dem Bildschirm erzeugen können und damit ein Prädikat `pprint`/1 zur Ausgabe von Bäumen definiert.
  - `write_canonical`/1 und `write`/1
  - `nl`/0 und `tab`/1
- Wir haben gesehen, wie Operatoren definiert werden und die zentralen Eigenschaften von Operatoren kennengelernt:
  - Typ
  - Präzedenz
  - Assoziativität
- **Keywords**: `functor`/3, `arg`/3, `=../2`, `pprint`/1, Operatoren
- **Ausblick Kapitel 10**: Cut und Negation

Übung: Matching- und Vergleichsoperatoren

Was antwortet Prolog auf die folgenden Anfragen?

```
1  ?- food(a) == food(a).
2  ?- food(a) =:= food(a).
3  ?- 3+4*5 == +(3,*(4,5)).
4  ?- 3+4*5 == *(+(3,4),5).
5  ?- [ha,hu] == '[|]'(ha,'[|]'(hu,[])).
6  ?- [ha,hu,ho] == [ha,hu|[ho]].
7  ?- [ha,X,ho] == [ha,hu|[ho]].
8  ?- [ha,X,ho] = [ha,hu|[ho]].
9  ?- (3<4) == <(3,4).
10 ?- 3+4*5 == X.
11 ?- 3+4*5 = X.
12 ?- 3+4*5 =:= X.
13 ?- 3+4*5 = X+Y.
14 ?- 3+4*5 = X*Y.
15 ?- 3+4*5 =:= 35.
16 ?- 3+4*5 =:= 23.
17 ?- 3+4*5 = 23.
18 ?- 6=X, X*5 =:= 30.
19 ?- 6=X, X == 6.
20 ?- 6=X, 7=X.
```

zurück

Übung: negierte Matching- und Vergleichsoperatoren

Was antwortet Prolog auf die folgenden Anfragen?

```
1  ?- a \= A.
2  ?- a \== A.
3  ?- a \= 'a'.
4  ?- food(a) \== food(b).
5  ?- 3+4*5 \= +(3,*(4,5)).
6  ?- 3+4*5 =\= *(+(3,4),5).
7  ?- [ha,X,ho] \== [ha,hu|[ho]].
8  ?- [ha,X,ho] \= [ha,hu|[ho]].
9  ?- 3+4*5 \== 23.
10 ?- 3+4*5 =\= 23.
11 ?- 6=X, X*5 =\= 20.
12 ?- 6=X, X*5 == 20.
13 ?- 6=X, 7\=X.
14 ?- (X is 3+4) \= is(8,3+4).
15 ?- X=b, (X==a) \== ==(b,a).
```

zurück

Übung: Typ von Termen

Was antwortet Prolog auf die folgenden Anfragen?

```
1  ?- atom(a).
2  ?- atom(7).
3  ?- atom(X).
4  ?- atomic(a).
5  ?- atomic(7).
6  ?- atomic(X).
7  ?- atomic(mag(eis)).
8  ?- var(X).
9  ?- var(a).
10 ?- X=a, nonvar(X).
11 ?- number(12.1234).
12 ?- number(3+4).
13 ?- integer(12.1234).
14 ?- number(2).
15 ?- X=a, var(X).
16 ?- var(X), X=a.
```

zurück

Übung: `functor/3`

Was antwortet Prolog auf die folgenden Anfragen?

```
?- functor(member(a,[a,b,c]),F,3).
?- functor(member(a,[a,b,c]),F,A).
?- functor(mag(popeye,eis),mag,2).
?- functor(4+5*6,F,A).
?- functor(4+5-6,F,A).
?- functor([a,b,c],F,A).
?- functor(C,hund,2).
?- functor(C,wizard(harry),2).
?- functor(C,7,2).
?- functor(X=a,F,A).
?- functor(4+5*6 < 4+5,F,A).
?- functor(X is 4+5*6,F,A).
```

zurück

Übung: `arg/3`

Was antwortet Prolog auf die folgenden Anfragen?

```
?- arg(2,[a,b,c],A).
?- arg(1,4+5*6,A).
?- arg(2,4+5-6).
?- arg(2,mag(popeye,eis),eis).
?- arg(1,[hu,ho,ha],A).
?- arg(X,mag(popeye,spinat),spinat).
?- arg(2,member(X,[a,b,c]),[_,_,Y]).
?- arg(3,[hu,ho,ha],A).
?- arg(2,C,hu).
```

zurück

Übung: univ `=../2`

Was antwortet Prolog auf die folgenden Anfragen?

```
?- X =.. [a,b,c].
?- [a,b,c] =.. X.
?- 4+5*6 =.. X.
?- X =.. 4+5*6.
?- X =.. [1,2,3].
?- X =.. [mag,popeye,essen(spinat)].
?- X =.. [+,2*4,3].
?- member(X,[a,b,c]) =.. X.
```

zurück

Übung: Typ von Termen

Schreiben sie ein zweistelliges Prädikat `termtype(+Term,?Type)`, das gelingt, wenn `Type` der Typ (atom, number, constant, variable etc.) des Terms `Term` ist. Hierbei sollen alle Typen, zu denen der Term gehört (beginnend mit dem spezifischsten) zurückgegeben werden:

```
?- termtype(Vincent,variable).
true.
?- termtype(mia,X).
X = atom ;
X = constant ;
X = simple_term ;
X = term ;
false.
?- termtype(dead(zed),X).
X = complex_term ;
X = term ;
false.
```

Übung: Parser

Ausgehend von der Grammatik in Kapitel 8, die einen Ableitungsbaum zu einem gegebenen String generiert,

1. schreiben sie ein Prädikat `parse/1`, das prüft, ob ein Satz von ihrer Grammatik generiert wird und wenn ja, den Ableitungsbaum mithilfe von `pprint/1` auf dem Bildschirm ausgibt.
2. Schreiben sie ein Prädikat `pprint_list/1`, das eine Liste als Argument nimmt und die Elemente der Liste nacheinander auf dem Bildschirm ausgibt ohne die Klammern und die Kommata der Liste.
3. Erweitern sie ihr Prädikat `parse/1` um `pprint_list/1`, so dass neben dem Ableitungsbaum auch der Satz ausgegeben wird.
4. Um nicht jedesmal wieder an der Konsole ganze Sätze eingeben zu müssen, schreiben sie nummerierte Beispielsätze in ihre Wissensbasis:

```
ex(1,[die,katze,jagt,eine,maus]).
ex(2,[die,katze,jagt,eine,maus,und,maeuse,klauen,katzen]).
```

Schreiben sie ein Prädikat `test/1`, das eine Zahl als Argument nimmt und den Beispielsatz mit der entsprechenden Nummer an ihr Prädikat `parse/1` weiterleitet.

Übung: Operatoren

Gegeben die folgenden Operatordefinitionen:

```
:- op(300, xfx, [are, is_a]).
:- op(300, fx, likes).
:- op(200, xfy, and).
:- op(100, fy, famous).
:- op(500, xf, or_not).
```

Welche der folgenden Ausdrücke sind wohlgeformt? Wie klammert Prolog die Ausdrücke intern?

```
1 ?- write_canonical(X is_a witch or_not).
2 ?- write_canonical(harry and ron and hermione are friends).
3 ?- write_canonical(harry is_a wizard and likes quidditch).
4 ?- write_canonical(dumbledore is_a famous famous wizard or_not).
5 ?- write_canonical(famous harry and ron are wizards).
6 ?- write_canonical(ron is_a wizards and harry likes quidditch).
```

zurück