Einführung in PROLOG VI

Terme vergleichen

PROLOG kennt ein wichtiges Prädikat zum Vergleichen von Termen, dass wir auch schon kennengelernt haben: ==
Zur Erinnerung sei noch einmal gesagt, dass es nicht dem = entspricht, dem Prädikat für Unifikation.
Beispiele:

```
?- a==a.
yes

?- a==b.
no

?- a=='a'.
yes

?- X==Y.
no

?- X=Y.
X = Y
Y = Y
yes

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

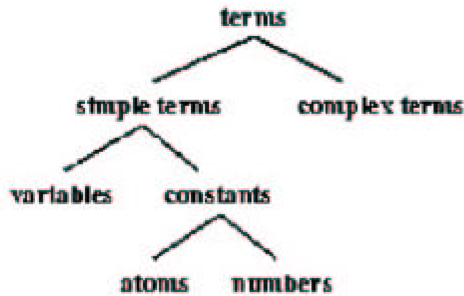
Es gibt auch ein Prädikat \== , welches so definiert ist, dass es genau dann erfüllt ist, wenn zwei Terme nicht identisch sind.

```
?- a\==a.
no

?-a\==b
yes

?-X\==a.
X=X
yes
```

Die letzte Query wollen wir uns genauer anschauen. Aus unseren Überlegungen oben, wie == mit uninstantiierten Variablen verfährt ist klar, dass **X==a** ein **no** ergeben würde, daher ergibt die untere Query ein yes und weißt weiter darauf hin, dass X weiterhin instantiiert werden muss.

Termtypen

Wir erinnern uns, dass wir am Anfang erklärten, welche Arten von Termen es bei PROLOG gibt: *Variablen* (variables), *Atome* (atoms) , *Zahlen* (numbers) und *Komplexe Terme* (complex terms). Außerdem kann man Zahlen und Atome unter dem Begriff *Konstanten* (constants) zusammenfassen, und Konstanten und Variablen bilden die *einfachen Terme* (simple terms). Das folgende Bild fasst es noch einmal zusammen:



PROLOG kennt einige eingebaute Prädikate. mit denen wir testen können, ob ein Term von einem gewissen Typ ist, was manchmal ganz nützlich sein kann:

atom/1	Testet, ob das Argument ein Atom ist
integer/1	Testet, ob das Argument eine ganze Zahl ist
float/1	Testet, ob das Argument ein Fließkommazahl ist
number/1	Testet, ob das Argument eine Zahl ist, also Integer oder Fließkomma
atomic/1	Testet, ob das Argument eine Konstante ist
var/1	Testet, ob das Argument uninstantiiert ist
nonvar/1	Testet, ob das Argument instantiiert ist

Beispiele:

```
?- atom(a).
yes

?-atom(lieben(student, vorlesung)).
no

?- atom(X).
no

?- X=a, atom(X).
yes

?-atom(X), X=a.
no
```

Die Struktur von Termen

Wenn uns ein komplexer Term vorliegt, was würde uns vor Allem interessieren, wenn wir nicht wissen, wie er aussieht? Wahrscheinlich wie der Prädikatsname (functor) lautet und welche Anzahl von Argumenten (arity) er hat. PROLOG bietet uns dafür das eingebaute Prädikat **functor/3** an.

```
?- functor(f(a,b), F, A).
F=f
A=2
yes

?- functor(a, F, A).
F=a
A=0
yes

?- functor(a=b, F, A).
F= =
A=2
yes

?- functor([a,b,c], X, Y).
X= .
Y= 2
yes
```

Die letzten beiden Queries müssen wir uns noch etwas genauer anschauen.
Wir erinnern uns, dass wir beim Verwenden des „=“ PROLOG-intern eigentlich das Prädikat =(a,b) verwenden, daher also die Aussage: Functor ist „=“, Arität ist 2.
Was wir aber bisher noch nicht wußten, ist dass PROLOG-intern Listen folgendermaßen abgespeichert werden: **.(Kopf, Rumpfliste),** das heißt die Query:

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

ergibt ein **yes.**
Wir können **functor** aber auch benutzen, um Terme zu konstruieren:

```
?- functor(T, f, 7).
```

ergibt zum Beispiel:

```
T = f(_1469, _1470, _1471, _1472, _1473, _1474, _1475)
yes
```

wobei _1469 in PROLOG für noch nicht instantiierte Variablen ohne Namen steht.
Damit können wir aber auch ein Prädikat konstruieren, das testet, ob etwas ein komplexer Term ist, was uns oben noch gefehlt hat:

```
complexterm(X):-
        nonvar(X),
        functor(X, _, A),
        A>0.
```

Passend dazu gibt es noch ein eingebautes Prädikat **arg/3**, welches uns Angaben über die Argumente von komplexen Termen macht. **arg(N, T, X).** benötigt eine Zahl **N** und einen komplexen Term **T** und instantiiert dann **X** mit dem *Nten* Argument von *T*
Beispiel:

?- arg(2, donaldsneffen(tick, trick, track), X).
X = trick
yes

Ein drittes nützliches Prädikat zur Analyse von Termstrukturen ist **=.. /2** Es benötigt einen komplexen Term und gibt dann eine Liste aus, die den Functor als erstes Element enthält und dann alle Argumente.Die Query **=.. (liebt(romeo, julia), X)** ergibt also **X=[liebt, romeo, julia].** Dieses Prädikat nennt man auch *univ* und es kann auch in der „inneren“ Schreibweise verwendet werden:

?- vater(draco, lucius) =.. X.
X = [vater, draco, lucius]
yes

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

?- erbe(Y, slytherin) =.. X.
Y = Y
X = [erbe, Y, slytherin]
yes

Univ (=..) ist immer nützlich, wenn irgendetwas mit allen Argumenten eines komplexen Terms gemacht werden muss. Da es die Argumente in Listenform ausgibt, können unsere normalen Prädikate zur Listenbearbeitung angewandt werden. Definieren wir zum Beispiel ein Prädikat **copy_term** , welches eine Kopie eines Terms erstellt und dabei alle Variablen, die im Term auftauchen, durch neue ersetzt. Die Kopie von *onkel(donald)* sollte also *onkel(donald)* sein, die Kopie von *eifersuechtig(donald,X)* sollte *eifersuechtig(donald, _G425)* sein, d.h X wurde durch eine neue Variable ersetzt.
Falls der input also eine Konstante ist, wird derselbe Term ausgegeben:

copyterm(X,X) :- atomic(X).

Falls der Input-Term eine Variable ist, soll die Kopie eine neue Varaiable sein:

copyterm(X, _):- var(X).

Wie geht es weiter mit komplexen Termen? Nun, copyterm sollte einen Term mit demselben Functor und derselben Arität ausgeben, und alle Argumente sollten Kopien der entsprechenden Argumente des Input-Terms sein. Das heißt, wir schauen uns alle Argumente des Input-Terms an und kopieren sie mit rekursiven Aufrufen von copyterm.
In PROLOG sieht das so aus:

```
copyterm(X,Y):-
        functor(X,F,A),
        functor(Y,F,A),
        A>0,
        X =.. [F|ArgsX],
        Y =.. [F|ArgsY],
        copy_terms_inlist(ArgsX,ArgsY).

copy_terms_inlist([],[]).

copy_terms_inlist([Kin|Rin],[Kout|Rout]):-
        copyterm(Kin,Kout),
        copy_terms_inlist(Rin,Rout).
```

Gehen Sie den Code Schritt für Schritt durch und verstehen Sie, was an welcher Stelle passiert.

Wir wiesen vorhin darauf hin, dass *univ* auch in der „inneren“ Schreibweise geschrieben werden kann. Korrekt hieße es, dass *univ* auch als **infix Operator** verwendet werden kann. Es gibt auch **prefix Operatoren**, die vor den Argumenten geschrieben werden und **postfix Operatoren** nach den Argumenten. Wir wollen uns damit im nächsten Kapitel näher beschäftigen:

Operatoren

PROLOG weiß, dass die Query *2+3*3* für *2+(3*3)* steht und nicht für *(2+3)*3*, also das Punktrechnung vor Strichrechnung geht. Aber warum?
PROLOG ordnet jedem Operator eine gewisse **Priorität** (precedence) zu. Die Priorität von + ist größer als die von * und damit wird + zum Hauptfunctor des Ausdrucks, denn PROLOGs interne Schreibweise +(2, *(3, 3)) ist eindeutig. Die Priorität von *is* ist noch höher, daher wird *11 is 2+3*3* als *is(11, +(2, *(3, 3)))* interpretiert und nicht als *+(is (11, 2), *(3, 3)))*. Priorität wird durch Zahlen angegeben, je höher die Zahl, desto höher die Priorität.

Außerdem hat jeder Operator noch eine **Assoziativität**. + ist **Links-Assoziativ**, was bedeutet, dass der Ausdruck auf der rechten Seite von + eine niedrigere Priorität haben muss, als + selbst. Die Operatoren ==, =:= und *is* hingegen sind **Nicht-Assoziativ**

Operatoren selbst definieren
PROLOG lässt uns Operatoren selbst definieren. Man könnte zum Beispiel einen postfix Operator macht_dick definieren und PROLOG würde einem erlauben cola macht_dick zu schreiben, anstelle von macht_dick(cola).

Definitionen von Operatoren sehen in PROLOG folgendermaßen aus:

```
:- op(Priorität, Typ, Name)
```

Die *Priorität* ist eine Zahl zwischen 1 und 1200. Die Priorität von = ist beispielsweise 700, die von + 500 und die von * 400. *Typ* spezifiziert den Typ und die Assoziativität. Im Falle von + ist das **yfx** , das steht für einen infix Operator, denn *x,y* repräsentieren die Argumente und *f* den Operator, der hier dazwischen steht. Zusätzlich bedeutet x ein Argument mit einer Priorität kleiner als die des Operators, und y ein Argument mit kleiner oder gleicher Priorität. Folgende Möglichkeiten gibt es also für Typ:

infix	*xfx, xfy, yfx*
prefix	*fx, fy*
postfix	*xf, yf*

Unser Operator könnte also folgendermaßen definiert sein:
:- op (500, xf, macht_dick)

Die Operatorendefinition spezifiert allerdings nicht die Bedeutung des Operators, sondern nur, wie man ihn syntaktisch einsetzen kann.

Nachtrag zur Assoziativität

Der Sinn von Priorität und Assoziativität besteht darin, die Interpretation eines Ausdruckes eindeutig zu machen, wenn er mehr als einen Operator enthält.
Der Operator mit der höchsten Priorität wird Hauptoperator, wie im Beispiel a + b * c, welches intern zuerst zu `+(a, *(b,c)) umgewandelt wird.`

Die Priorität eines Atoms ist immer 0. Die Priorität eines Sub-Ausdrucks ist die seines Hauptoperators. Die einzige Methode die Priorität eines Ausdrucks zu ändern, ist ihn einzuklammern, denn jeder eingeklammerte Ausdruck hat Priorität 0.

Erst wenn die Priorität der Operatoren eines Ausdrucks gleich ist, kommt die Assoziativität ins Spiel!
Wir erinnern uns: **f** für den Operator, **x** für einen Operator mit immer kleinerer Priorität, **y** für einen mit kleinerer oder gleicher Priorität.

Schauen wir uns dazu ein paar Beispiele für Infix-Operatoren an:

Ein rechtsassoziativer Operator ++
:- op(100, xfy, ++).

Damit wird der Ausdruck:
a ++ b ++ c
folgendermaßen abgearbeitet:
a ++ (b ++ c)

Ein linksassozitiver Operator --
:- op(100, yfx, --).

Damit wird der Ausdruck:
a -- b -- c
so abgearbeitet:
(a -- b) -- c

Ein nichtassoziativer Operator **
:- op(100, xfx, **).

Damit ist der Ausdruck
a ** b ** c
illegal und benötigt Klammern.

Prefix und Postfix operatoren.

:- op(100, fy, not).

Damit wird der Ausdruck:

not not P

folgendermaßen abgearbeitet

not(not P)

Würden wir ‘not’ mit dem dem Typ 'fx' deklarieren, wäre nur die untere Schreibweise legal.

Beispiele für eingebaute Operatoren in PROLOG:

```
:- op( 1200, xfx, [ :-, -> ]).
:- op( 1200,  fx, [ :-, ?- ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1000, xfy, [ ',' ]).
:- op(  700, xfx, [ =, is, =.., ==, \==,
                    =:=, =\=, <, >, =<, >= ]).
:- op(  500, yfx, [ +, -]).
:- op(  500,  fx, [ +, - ]).
:- op(  300, xfx, [ mod ]).
:- op(  200, xfy, [ ^ ]).
```

Übungen

Übung 6.1

Schreiben Sie ein zweistelliges Prädikat termtype(+Term,?Type) , welches für einen Term die möglichen Typen des Terms anzeigt (atom, zahl, konstante, variable usw.).
Die Typen sollten in der aufsteigender Reihenfolge angegeben werden, z.B.

?-termtype(Vincent,variable).
yes

?-termtype(mia,X).
X = atom ;
X = constant ;
X = simple_term ;
X = term ;
no

?-termtype(dead(zed),X).
X = complex_term ;
X = term ;
no

Übung 6.2

Nehmen wir an, wir hätten folgende Operatordefinitionen:

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

Welche der folgenden Terme sind wohlgeformt?

```
?-X is_a witch.
?-harry and ron and hermione are friends.
?-harry is_a wizard and likes quidditch.
?-dumbledore is_a famous famous wizard.
?-harry likes likes quiditch
?-dumbledore is_a harry is_a famous wizard
```

Übung 6.3

Schreiben Sie ein Programm, das das Prädikat groundterm(+Term) definiert, um zu testen, ob ein Term ein Groundterm ist, also ein Term der keine Variablen enthält.

?-groundterm(X).
no

?-groundterm(neffe(dagobert, donald)).
yes

?-groundterm(neffe(donald,X)).
no