

Logische und funktionale Programmierung

Vorlesung 8: Arithmetik, Listenprädikate, weitere Prolog Prädikate

Babeş-Bolyai Universität, Department für Informatik, Cluj-Napoca
csacarea@cs.ubbcluj.ro



ARITHMETIK IN PROLOG

- Die meisten Prologimplementierungen stellen Operatoren zur Verarbeitung von Zahlen zur Verfügung.
- Hierzu gehören die arithmetischen Operatoren + (Addition), - (Subtraktion), * (Multiplikation), / (Division), // (ganzzahlige Division), mod (modulo) und ^ (Exponent).
- Alle Operatoren können auch als Funktoren verwendet werden: Statt $3+4$ kann man auch $+(3,4)$ schreiben.
- Die verwendeten Symbole für die Operatoren hängen von dem jeweiligen Prolog-Interpreter ab (hier angegeben für SWI-Prolog).
- Vorsicht: Arithmetische Operationen gehören nicht zu den Kernkonzepten von Prolog. Mit ihnen verlässt man das auf Unifikation basierende Grundprinzip der deklarativen Programmierung.



RECHNEN IN PROLOG

```
?- X is 3+4.
```

```
X = 7.
```

```
?- X is 3*4.
```

```
X = 12.
```

```
?- X is 3/4.
```

```
X = 0.75.
```

```
?- X is 13 mod 5.
```

```
X = 3.
```

% Prolog beherrscht Punkt- vor Strichrechnung:

```
?- X is 3+4*5.
```

```
X = 23.
```

% Klammern koennen wie ueblich verwendet werden:

```
?- X is (3+4)*5.
```

```
X = 35.
```



ARITHMETISCHE OPERATOREN UND DIE EVALUATION

- Arithmetische Ausdrücke werden in Prolog nicht evaluiert bzw. ausgewertet, sondern sind gewöhnliche zusammengesetzte Terme.

```
?- X = 2 + 3.  
X = 2+3.  
?- 2+3 = 2+3.  
true.  
?- 2+3 = +(2,3).  
true.
```

- Um arithmetische Ausdrücke in Prolog zu berechnen benötigt man den Infix-Operator **is**.

```
?- X is 2 + 3.  
X = 5  
?- is(X,2*3).  
X = 6.
```



DER EVALUATIONSOOPERATOR IS/2

Vorsicht, da der Evaluationsoperator `is/2` außerhalb der normalen Programmlogik von Prolog steht, stellt er besondere Ansprüche:

- Der Evaluationsoperator `is/2` erzwingt die sofortige Auswertung des zweiten Arguments,
- daher muss das zweite Argument ein evaluierbarer arithmetischer Ausdruck sein:

```
?- X is 3+5.  
X = 8.  
?- 3+5 is X.  
ERROR: is/2: Arguments are not sufficiently instantiated  
?- X is 4+Y.  
ERROR: is/2: Arguments are not sufficiently instantiated  
?- X is a.  
ERROR: Arithmetic: 'a' is not a function
```

- Ist das zweite Argument nicht evaluierbar, so bricht Prolog mit einer Fehlermeldung ab.



VERGLEICH IS/2 MIT NORMALEN PROLOGPRÄDIKATEN

Der Evaluationsoperator `is/2` unterscheidet sich grundlegend von "normalen" Prologprädikaten wie `member/2`.

Werden "normale" Prologprädikate "falsch" instantiiert, kommt es zu keinem Programmabbruch. Die Aussage kann lediglich nicht bewiesen werden:

```
?- member(a,b).  
false.  
?- member([a,b],a).  
false.  
?- X is a.  
ERROR: Arithmetic: 'a' is not a function
```



ARITHMETISCHE VERGLEICHOPERATOREN

Neben dem Evaluationsoperator `is/2` gibt es weitere Operatoren, die das Evaluieren arithmetischer Ausdrücke erzwingen.

Die zweistelligen **Vergleichsoperatoren** `<` (kleiner), `=<`, (kleiner gleich), `>` (größer), `=>` (größer gleich), `=:=` (gleich) und `=\=` (ungleich) erzwingen die sofortige Evaluation beider Argumente.

```
?- 1+4 < 3*5.  
true.  
?- 1+7 =< 3*2.  
false.  
?- 1+3 =:= 2*2.  
true.  
?- 1+3 =\= 2*3.  
true.  
?- X < 3.  
ERROR: </2: Arguments are not sufficiently instantiated  
?- 3 =:= 2+X.  
ERROR: =:/2: Arguments are not sufficiently instantiated
```



EVALUATION ERZWINGENDE OPERATOREN IN PRÄDIKATSDEFINITIONEN

Evaluation erzwingende Operatoren können in Prädikatsdefinitionen eingesetzt werden.

Allerdings muss sichergestellt werden, dass beim Aufruf des Prädikats die zu evaluierenden Ausdrücke vollständig instantiiert sind.

```
% Definition
double_and_add3(X,Y):- Y is 2*X + 3.

% Aufrufe:
?- double_and_add3(3,9).
true.

?- double_and_add3(4,Y).
X=11.

?- double_and_add3(X,11).
ERROR: is/2: Arguments are not sufficiently instantiated
```



LISTENLÄNGE BESTIMMEN OHNE AKKUMULATOR

Die Länge einer Liste ist die Anzahl ihrer Elemente. Z.B. hat die Liste [a,b,b,a] die Länge 4.

rekursive Längendefinition

- ① Die leere Liste hat die Länge 0.
- ② Eine nichtleere Liste hat eine Länge, die um 1 höher ist als die Länge ihres Tails.

```
% len1/2
% len1(List, Length)
len1([],0).
len1([_|T],N):-
    len1(T,X),
    N is X+1.
```

```
?- len1([a,[b,e,[f,g]],food(cheese),X],4).
true.
?- len1([a,b,a],X).
X=3.
```



TRACE: LISTENLÄNGE OHNE AKKUMULATOR

Prädikatsdefinition:

```
% len1/2
% len1(List, Length)
len1([], 0).
len1([_|T], N) :-  
    len1(T, X),
    N is X+1.
```

trace einer Beispielenfrage:

```
?- len1([a,a,a], Len).
Call: (7) len1([a,a,a], _X1) ?
Call: (8) len1([a,a], _X2) ?
Call: (9) len1([a], _X3) ?
Call: (10) len1([], _X4) ?
Exit: (10) len1([], 0) ?
Call: (10) _X3 is 0+1 ?
Exit: (10) 1 is 0+1 ?
Call: (9) _X2 is 1+1 ?
Exit: (9) 2 is 1+1 ?
Call: (8) _X1 is 2+1 ?
Exit: (8) 3 is 2+1 ?
Exit: (7) len1([a,a,a], 3) ?
Len = 3.
```



LISTENLÄNGE BESTIMMEN MIT AKKUMULATOR

- Akkumulatoren (*accumulators*) dienen dem Aufsammeln von Zwischenergebnissen.
- Akkumulatoren ermöglichen eine effizientere Implementierung in Prolog, da Variablen früher instantiiert werden können.
- Rekursive Programmierung mit Akkumulatoren zählt zu den zentralen Programmiertechniken in Prolog.

```
% len2/2
% len2(List, Length)
len2(List,Length) :- accLen(List,0,Length).

% accLen/3
% accLen(List,Accumulator,Length)
accLen([_|T],Acc,Length) :-
    NewAcc is Acc+1,
    accLen(T,NewAcc,Length).
accLen([],Length,Length).
```



TRACE: LISTENLÄNGE MIT AKKUMULATOR

Prädikatsdefinition:

```
% len2/2
% len2(List, Length)
len2(List, Length) :-
    accLen(List, 0, Length).

% accLen/3
% accLen(List, Acc, Length)
accLen([_|T], Acc, L) :-
    NewAcc is Acc+1,
    accLen(T, NewAcc, L).
accLen([], Acc, Acc).
```

trace einer Beispielenfrage:

```
?- len2([a,a,a],Len).
Call: (7) len2([a,a,a], _X1) ?
Call: (8) accLen([a,a,a], 0, _X1) ?
Call: (9) _X is 0+1 ?
Exit: (9) 1 is 0+1 ?
Call: (9) accLen([a,a], 1, _X1) ?
Call: (10) _X is 1+1 ?
Exit: (10) 2 is 1+1 ?
Call: (10) accLen([a], 2, _X1) ?
Call: (11) _X is 2+1 ?
Exit: (11) 3 is 2+1 ?
Call: (11) accLen([], 3, _X1) ?
Exit: (11) accLen([], 3, 3) ?
Exit: (10) accLen([a], 2, 3) ?
Exit: (9) accLen([a,a], 1, 3) ?
Exit: (8) accLen([a,a,a], 0, 3) ?
Exit: (7) len2([a,a,a], 3) ?
Len = 3.
```



VERGLEICH LÄNGE MIT UND OHNE AKKUMULATOR

ohne Akkumulator:

```
?- len1([a,a,a],Len).
Call: (7) len1([a,a,a], _X1) ?
Call: (8) len1([a,a], _X2) ?
Call: (9) len1([a], _X3) ?
Call: (10) len1([], _X4) ?
Exit: (10) len1([], 0) ?
Call: (10) _X3 is 0+1 ?
Exit: (10) 1 is 0+1 ?
Exit: (9) len1([a], 1) ?
Call: (9) _X2 is 1+1 ?
Exit: (9) 2 is 1+1 ?
Exit: (8) len1([a,a], 2) ?
Call: (8) _X1 is 2+1 ?
Exit: (8) 3 is 2+1 ?
Exit: (7) len1([a,a,a], 3) ?
Len = 3.
```

mit Akkumulator

```
?- len2([a,a,a],Len).
Call: (7) len2([a,a,a], _X1) ?
Call: (8) accLen([a,a,a], 0, _X1) ?
Call: (9) _X is 0+1 ?
Exit: (9) 1 is 0+1 ?
Call: (9) accLen([a,a], 1, _X1) ?
Call: (10) _X is 1+1 ?
Exit: (10) 2 is 1+1 ?
Call: (10) accLen([a], 2, _X1) ?
Call: (11) _X is 2+1 ?
Exit: (11) 3 is 2+1 ?
Call: (11) accLen([], 3, _X1) ?
Exit: (11) accLen([], 3, 3) ?
Exit: (10) accLen([a], 2, 3) ?
Exit: (9) accLen([a,a], 1, 3) ?
Exit: (8) accLen([a,a,a], 0, 3) ?
Exit: (7) len2([a,a,a], 3) ?
Len = 3.
```



MAXIMALES LISTENELEMENT BESTIMMEN MIT AKKUMULATOR

```
1 % max1/2
2 % max1(List, ListMax)
3 max1([H|T], Max) :-  
    accMax(T, H, Max).
4
5
6 % accMax/3
7 % accMax(List, Accum., ListMax)
8 accMax([], Max, Max).
9
10 accMax([H|T], Acc, Max) :-  
    H > Acc,  
    accMax(T, H, Max).
11
12 accMax([H|T], Acc, Max) :-  
    H =< Acc,  
    accMax(T, Acc, Max).
```

Grundidee: Die Liste wird von vorne nach hinten rekursiv aufgespalten. Die Variable `Acc` fasst das jeweils bis dato höchste Listenelement.

Zeile 4: Zu Beginn ist der Kopf der Liste das höchste bis dato gesehene Listenelement.

Zeile 10-12: Ist der Kopf der aktuellen Liste größer als das bisherige Maximum, das im Akkumulator gespeichert ist, wird der Akkumulator durch den Kopf ersetzt.

Zeile 14-16: Ist der Kopf der aktuellen Liste nicht größer als das bisherige Maximum, das im Akkumulator gespeichert ist, bleibt der Akkumulator erhalten.

Zeile 8: Ist die Liste abgearbeitet, speichert der Akkumulator das maximale Listenelement.



MAXIMALES LISTENELEMENT BESTIMMEN OHNE AKKUMULATOR

```
1  % max2/2 bestimmt das maximale
2  % Listenelement einer Liste
3  % mit nicht negativen Zahlen
4  % max2(List,ListMax)
5
6  max2([H|T],H) :-           % Zeile 5-7
7      max2(T,MaxT),
8      H > MaxT.
9
10 max2([H|T],MaxT) :-          % Zeile 9-11
11     max2(T,MaxT),
12     H =< MaxT.
13
14 max2([],0).
```

Zeile 5-7: Der Kopf einer Liste ist das Maximum der gesamten Liste, wenn er größer ist als das Maximum der Restliste.

Zeile 9-11: Ist der Kopf der Liste nicht größer als das Maximum der Restliste, dann ist das Maximum der Restliste das Maximum der gesamten Liste.

Zeile 13: Per Definition erklären wir, dass die leere Liste das Maximum 0 hat.



AKKUMULATOREN: STRUKTUR DER PROGRAMME

Listenverarbeitung ohne Akkumulator

- Die eigentliche Verarbeitung beginnt am tiefsten Punkt der Rekursion.
- Die initiale Instanziierung der Lösungsvariable erfolgt am tiefsten Punkt der Rekursion.
- In jedem Schritt aus der Rekursion heraus erfolgt ein Verarbeitungsschritt.

```
p([H|T],Sol) :-  
    p(T,NewSol),  
    ...,  
    Sol is ... NewSol ...,  
    ...  
    .  
  
p([],Initial).
```

Listenverarbeitung mit Akkumulator

- Die Instanziierung der Akkumulatorvariable erfolgt beim ersten Aufruf.
- Am tiefsten Punkt der Rekursion wird die Lösungsvariable mit dem Akkumulator unifiziert.
- In jedem Schritt in die Rekursion hinein erfolgt ein Verarbeitungsschritt.

```
p(List,Sol) :- p(List,InitialAcc,Sol).  
  
p([H|T],Acc,Sol) :-  
    NewAcc is ... Acc ...,  
    ...,  
    p(T,NewAcc,Sol),  
    ...  
    .  
  
p([],Sol,Sol).
```



ZUSAMMENFASSUNG

- Keywords: Rechnen in Prolog mit dem Evaluationsoperator `is`, arithmetische Vergleichsoperatoren, Akkumulatoren.
- Wichtig: Die rekursive Verarbeitung von Listen mit Akkumulatoren ist eine zentrale Programmiertechnik in Prolog.
- Vorsicht: Die arithmetischen Vergleichsoperatoren und der Operator `is` fordern zwingend sofort evaluierbare Terme. Uninstantiierte Terme führen zu einem Abbruch mit Fehlermeldung.



WIEDERHOLUNG

Vier Prädikate zur rekursiven Listenverarbeitung demonstrieren Basistechniken für beliebig komplexe Operationen auf Listen:

- **member**/2 Zugriff auf Listenelemente.

`member(Element, List)`

- **append**/3 Konkatenation von Listen.

`append(List1, List2, Konkatlist)`

- **delete**/3 Löschen/Einfügen in Listen.

`delete(Element, List, ListDeleted)`

- **reverse**/2 Umkehren von Listen.

`reverse(List, ListReversed)`



WIEDERHOLUNG

KONKATENATION VON LISTEN: APPEND/3

```
% append/3
% append(L1,L2,L3)
% L3 is the result of concatenating L1 and L2
% L1 o L2 = L3

append([],L,L).

append([H|T1],L2,[H|T3]) :-
    append(T1,L2,T3).
```

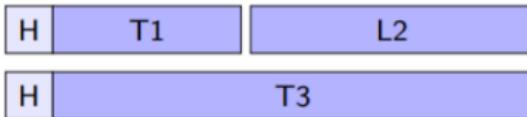
```
?- append([1,2,3],[4,5,6],[1,2,3,4,5,6]).  
true.  
?- append([1,2,3],[4,5,6],L).  
L = [1,2,3,4,5,6].  
?- append(L,[4,5,6],[1,2,3,4,5,6]).  
L = [1,2,3].  
?- append([1,2,3],L,[1,2,3,4,5,6]).  
L = [4,5,6].
```



APPEND / 3 DEKLARATIV

```
append([], L, L).  
  
append([H|T1], L2, [H|T3]) :-  
    append(T1, L2, T3).
```

- Die Konkatenation einer Liste L an die leere Liste liefert L als Ergebnis.
- Wenn die Konkatenation der Listen L1 und L2 die Liste L3 ergibt, dann ergibt die Konkatenation der um ein zusätzliches Kopfelement H erweiterten List L1 mit L2 die um denselben Kopf erweiterte Liste L3.



$$[H|T1] \circ L2 = [H|T3] \text{ wenn } T1 \circ L2 = T3$$



APPEND / 3 PROZEDURAL

```
append([],L,L).  
  
append([H|T1],L2,[H|T3]) :-  
    append(T1,L2,T3).
```

```
?- append([1,2,3],[4,5,6],L).  
Call: (7)  append([1,2,3], [4,5,6], _G2304) ?  
Call: (8)  append([2,3], [4,5,6], _G2392) ?  
Call: (9)  append([3], [4,5,6], _G2395) ?  
Call: (10) append([], [4,5,6], _G2398) ?  
Exit: (10) append([], [4,5,6], [4,5,6]) ?  
Exit: (9)  append([3], [4,5,6], [3,4,5,6]) ?  
Exit: (8)  append([2,3], [4,5,6], [2,3,4,5,6]) ?  
Exit: (7)  append([1,2,3], [4,5,6], [1,2,3,4,5,6]) ?  
L = [1,2,3,4,5,6] ;
```



VERWENDUNG VON APPEND/3

Das Prädikat `append/3` kann

- **testen**, ob eine Liste die Konkatenation von zwei Listen ist:
`append(+,+,+):`
`append([a,b,c],[x,y,z],[a,b,c,x,y,z]).`
- zwei Listen **konkatenieren**: `append(+,+,-):`
`append([a,b,c],[x,y,z],L).`
- Listen **zerlegen**: `append(-,-,+)`, `append(-,+,+)`, `append(+,-,+):`
`append(X,Y,[a,b,c]).`
`append(X,[b,c,d],[a,b,c,d]).`
`append([a,b],X,[a,b,c,d]).`



BESONDERHEITEN VON APPEND/3

Mit dem Prädikat **append**/3 können sehr unterschiedliche Funktionen implementiert werden. Dennoch muss man beachten, dass

- bei jedem Aufruf von **append**/3 die Liste im ersten Argument komplett abgearbeitet werden muss.
- aufgrund der kompletten Listenarbeitung Programme mit vielen Aufrufen von **append**/3 sehr schnell ineffizient werden können.

Man sollte also bei der Verwendung von **append**/3 in rekursiven Prädikaten vorsichtig sein.



SUFFIXE, PRÄFIXE UND ALLGEMEINE SUBLISTEN: PREFIX/2, SUFFIX/2, SUBLIST/2

Das Prädikat `append/3` kann für die Definition von Sublisten eingesetzt werden:

- **Präfixe** der Liste `[a,b,c,d]`: `[]`, `[a]`, `[a,b]`, `[a,b,c]`, `[a,b,c,d]`

```
prefix(P,L) :- append(P,_,L).
```

- **Suffixe** der Liste `[a,b,c,d]`: `[]`, `[d]`, `[c,d]`, `[b,c,d]`, `[a,b,c,d]`

```
suffix(S,L) :- append( _,S,L).
```

- **Sublisten** der Liste `[a,b,c]`: `[]`, `[a]`, `[a,b]`, `[a,b,c]`, `[b]`, `[b,c]`, `[c]`

```
sublist(SL, L) :- prefix(P,L), suffix(SL,P).
```

L



SUFFIXE, PRÄFIXE UND ALLGEMEINE SUBLISTEN: PREFIX/2, SUFFIX/2, SUBLIST/2

Das Prädikat `append/3` kann für die Definition von Sublisten eingesetzt werden:

- **Präfixe** der Liste `[a,b,c,d]`: `[]`, `[a]`, `[a,b]`, `[a,b,c]`, `[a,b,c,d]`

```
prefix(P,L) :- append(P,_,L).
```

- **Suffixe** der Liste `[a,b,c,d]`: `[]`, `[d]`, `[c,d]`, `[b,c,d]`, `[a,b,c,d]`

```
suffix(S,L) :- append( _,S,L).
```

- **Sublisten** der Liste `[a,b,c]`: `[]`, `[a]`, `[a,b]`, `[a,b,c]`, `[b]`, `[b,c]`, `[c]`

```
sublist(SL, L) :- prefix(P,L), suffix(SL,P).
```



SUFFIXE, PRÄFIXE UND ALLGEMEINE SUBLISTEN: PREFIX/2, SUFFIX/2, SUBLIST/2

Das Prädikat `append/3` kann für die Definition von Sublisten eingesetzt werden:

- **Präfixe** der Liste `[a,b,c,d]`: `[]`, `[a]`, `[a,b]`, `[a,b,c]`, `[a,b,c,d]`

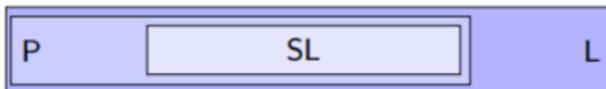
```
prefix(P,L) :- append(P,_,L).
```

- **Suffixe** der Liste `[a,b,c,d]`: `[]`, `[d]`, `[c,d]`, `[b,c,d]`, `[a,b,c,d]`

```
suffix(S,L) :- append( _,S,L).
```

- **Sublisten** der Liste `[a,b,c]`: `[]`, `[a]`, `[a,b]`, `[a,b,c]`, `[b]`, `[b,c]`, `[c]`

```
sublist(SL, L) :- prefix(P,L), suffix(SL,P).
```



LÖSCHEN EINES ELEMENTS: DELETE / 3

```
% delete/3
% delete(Term, Liste1, Liste2)
delete(X, [X|T], T).
delete(X, [H|T1], [H|T2]) :-
    delete(X, T1, T2).
```

`delete/3` setzt einen Term und zwei Listen derart in Beziehung, daß `Liste2` das Ergebnis des einmaligen Löschens von `Term` an einer beliebigen Position in `Liste1` repräsentiert.

```
?- delete(b,[a,b,c],[a,c]).  
true.  
?- delete(c,[a,b,c],X).  
X=[a,b]  
?- delete(X,[a,b,c,d],[a,b,d]).  
X = c  
?- delete(1,X,[a,b,c]).  
X = [1, a, b, c] ;  
X = [a, 1, b, c] ;
```



UMDREHEN VON LISTEN: NAIVEREV / 2 (NAIVE DEFINITION)

Zwei Listen sind zueinander **revers**, wenn die eine Liste gleich der anderen ist, wenn man die Reihenfolge der Elemente umdreht.

```
naiverev([], []).  
naiverev([H|T], R) :-  
    naiverev(T, RevT),  
    append(RevT, [H], R).
```

Deklarative Idee: Die Umkehrung einer nichtleeren Liste $[H|T]$ ergibt sich, indem man an die Umkehrung von T eine Liste mit dem Kopf H als einzigem Element konkateniert.

```
?- naiverev([a,b,c],[c,b,a]).  
true.  
?- naiverev([1,[2,3]],X).  
X=[[2,3],1].  
?- naiverev(X,[a,b,c]).  
X=[c,b,a].  
?- naiverev([],X).  
X=[].
```



WARUM NAIVES REVERSE?

- Das naive naiverev/2 wird naiv genannt, weil das zu lösende Problem eigentlich mit linearer Laufzeit gelöst werden könnte.
- Das naive naiverev/2 benötigt jedoch durch den Einsatz von append/3 kubische Laufzeit.
- Betrachte den Trace von naiverev([a,b,c,d],X).



REVERSE / 2 MIT AKKUMULATOR

```
% reverse/2
% reverse(Liste, UmgekehrteListe)
reverse(L, RL) :- reverse(L, [], RL).

% reverse/3
% reverse(Liste, Akkumulator, UmgekehrteListe)
reverse([], RL, RL).
reverse([H|T], RT, RL) :-
    reverse(T, [H|RT], RL).
```

Deklarative Idee:

- Die Elemente der ersten Liste werden nacheinander auf einen neuen Stapel gelegt (den Akkumulator, der als leere Liste startet).
- In jedem Schritt schrumpft die erste Liste und wächst der Akkumulator um ein Element.
- Die Elemente aus der ersten Liste geraten im Akkumulator in umgekehrte Reihenfolge (das erste Element kommt als erstes in den Akkumulator und damit an dessen letzten Platz).
- Wenn die erste Liste leer ist, liefert der Akkumulator das Ergebnis.



REVERSE / 2 PROZEDURAL

```
?- reverse([a,b,c,d],X).
Call: (7)  reverse([a,b,c,d],           _G2273) ?
Call: (8)  reverse([a,b,c,d],[],        _G2273) ?
Call: (9)  reverse([b,c,d],[a],        _G2273) ?
Call: (10) reverse([c,d],[b,a],       _G2273) ?
Call: (11) reverse([d],[c,b,a],       _G2273) ?
Call: (12) reverse([], [d,c,b,a],     _G2273) ?
Exit: (12) reverse([], [d,c,b,a], [d,c,b,a]) ?
Exit: (11) reverse([d],[c,b,a], [d,c,b,a]) ?
Exit: (10) reverse([c,d],[b,a], [d,c,b,a]) ?
Exit: (9)  reverse([b,c,d],[a], [d,c,b,a]) ?
Exit: (8)  reverse([a,b,c,d],[], [d,c,b,a]) ?
Exit: (7)  reverse([a,b,c,d], [d,c,b,a]) ?
X = [d,c,b,a]
```



LISTENVERARBEITUNG MIT AKKUMULATORLISTE

- Tail des Akkumulators steht im Kopf der Regel;
- Zerlegung der Akkumulatorliste erfolgt im rekursiven Aufruf;
- Akkumulatorliste wird mit Rekursion länger.

```
p(...[H|T],TAcc,...):-  
    ...,  
    p(...,T,[H|TAcc],...),  
    ...
```



DIFFERENZLISTEN

Listen können auch als **Differenzlisten** repräsentiert werden:

- Eine Differenzliste ist ein Paar von Listen (L_1, L_2), wobei L_2 ein Suffix von L_1 repräsentiert.
- Die Elemente der Differenzliste sind die nach Abzug von Suffix L_2 verbleibenden Elemente in L_1 .

Differenzliste	gewöhnliche Liste
$([E_1, \dots, E_n] \mid T), T)$	$[E_1, \dots, E_n]$
$(L, [])$	L
(L, L)	$[]$



BEISPIEL: [1,2,3] ALS DIFFERENZLISTE

Für jede gewöhnliche Liste gibt es unendlich viele Darstellungen als Differenzliste:

```
% [1,2,3] als Differenzliste:
```

```
([1,2,3],[])
([1,2,3,4],[4])
([1,2,3,4,5],[4,5])
([1,2,3,4,5,6],[4,5,6])
([1,2,3,4,5,6,7],[4,5,6,7])
...
([1,2,3|T],T)
([1,2,3,a|T],[a|T])
([1,2,3,a,b|T],[a,b|T])
...
```



VORTEIL VON DIFFERENZLISTEN: KONKATENATION IN EINEM SCHRITT

Differenzlisten können in einem Schritt konkateniert werden:

```
% append_dl/3
% append_dl(DiffList1, DiffList2, DiffList3)
append_dl((A,B),(B,C),(A,C)).
```



VORTEIL VON DIFFERENZLISTEN: KONKATENATION IN EINEM SCHRITT

Differenzlisten können in einem Schritt konkateniert werden:

```
% append_dl/3
% append_dl(DiffList1, DiffList2, DiffList3)
append_dl((A,B),(B,C),(A,C)).
```

A

B

(A,B)



VORTEIL VON DIFFERENZLISTEN: KONKATENATION IN EINEM SCHRITT

Differenzlisten können in einem Schritt konkateniert werden:

```
% append_dl/3
% append_dl(DiffList1, DiffList2, DiffList3)
append_dl((A,B),(B,C),(A,C)).
```



VORTEIL VON DIFFERENZLISTEN: KONKATENATION IN EINEM SCHRITT

Differenzlisten können in einem Schritt konkateniert werden:

```
% append_dl/3
% append_dl(DiffList1, DiffList2, DiffList3)
append_dl((A,B),(B,C),(A,C)).
```

A

C

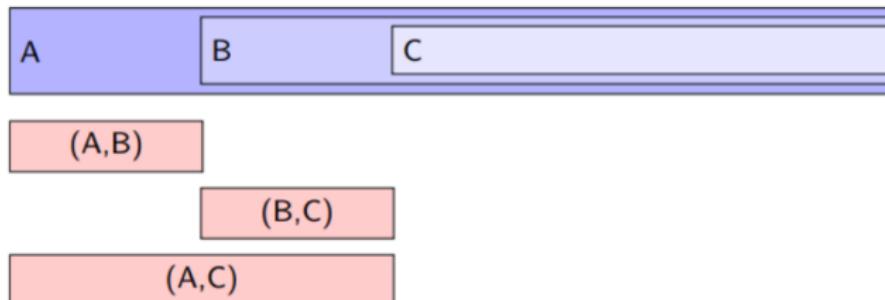
(A,C)



VORTEIL VON DIFFERENZLISTEN: KONKATENATION IN EINEM SCHRITT

Differenzlisten können in einem Schritt konkateniert werden:

```
% append_dl/3
% append_dl(DiffList1, DiffList2, DiffList3)
append_dl((A,B),(B,C),(A,C)).
```



VORTEIL VON DIFFERENZLISTEN: KONKATENATION IN EINEM SCHRITT

Differenzlisten können in einem Schritt konkateniert werden:

```
% append_dl/3
% append_dl(DiffList1, DiffList2, DiffList3)
append_dl((A,B),(B,C),(A,C)).
```



VORTEIL VON DIFFERENZLISTEN: KONKATENATION IN EINEM SCHRITT

Differenzlisten können in einem Schritt konkateniert werden:

```
% append_dl/3
% append_dl(DiffList1, DiffList2, DiffList3)
append_dl((A,B),(B,C),(A,C)).
```

Arbeitsweise von `append_dl/3`:

```
?- D1 = ([1,2,3|T1],T1),
   D2 = ([4,5,6|T2],T2),
   append_dl(D1,D2,D3).
D3 = ([1,2,3,4,5,6|T2],T2)
```



VORTEIL VON DIFFERENZLISTEN: KONKATENATION IN EINEM SCHRITT

Differenzlisten können in einem Schritt konkateniert werden:

```
% append_dl/3
% append_dl(DiffList1, DiffList2, DiffList3)
append_dl((A,B),(B,C),(A,C)).
```

Arbeitsweise von `append_dl/3`:

```
?- D1 = ([1,2,3|T1],T1),
D2 = ([4,5,6|T2],T2),
append_dl(D1,D2,D3).
D3 = ([1,2,3,4,5,6|T2],T2)
```

```
?- append_dl(      D1          ,      D2          , D3 ) .
?- append_dl(([1,2,3|T1],T1),([4,5,6|T2],T2), D3 ) .
append_dl((      A          , B),(      B          , C),(A,C)).
```

```
A = [1,2,3|T1]
B = T1 = [4,5,6|T2]
C = T2
```

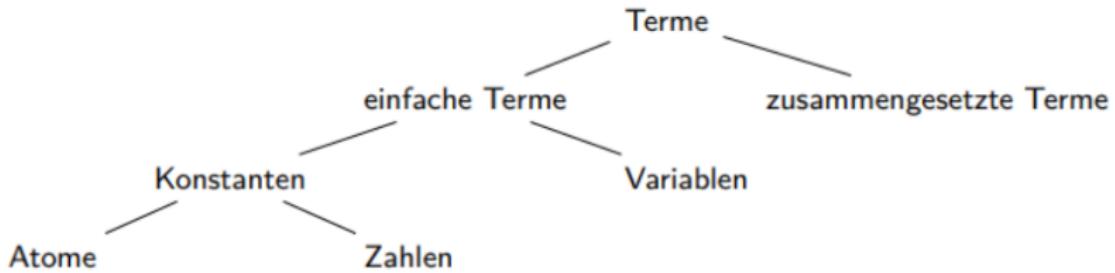
```
=> A = [1,2,3|[4,5,6|T2]] = [1,2,3,4,5,6|T2]
=> D3 = (A,C) = ([1,2,3,4,5,6|T2],T2)
```



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.



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- / UNIFIKATIONSOOPERATOR

Der Matchingoperator „=“ gelingt, wenn Funktor und Stelligkeit übereinstimmen und 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 Ungleichheitsoperator „`=\=`“ 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.
```



BERSICHT MATCHING- UND VERGLEICHSOPERATOREN

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



ANALYSE VON NICHT ZUSAMMENGESETZTEN TERMEN

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

Prädikat	Funktion
<code>atom/1</code>	Testet ob das Argument ein Atom ist
<code>integer/1</code>	Testet ob das Argument eine natürliche Zahl ist
<code>number/1</code>	Testet ob das Argument eine Zahl ist
<code>atomic/1</code>	Testet ob das Argument eine Konstante ist
<code>var/1</code>	Testet ob das Argument eine uninstantiierte Variable ist
<code>nonvar/1</code>	Testet ob das Argument keine uninstantiierte Variable ist

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

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



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
```



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 grösser 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.
```



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].
```



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
```



OPERATOREN EXTERNE UND INTERNE NOTATION

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

interne Notation	nutzerfreundliche externe Notation
$+(1,2)$	$1+2$
$\text{is}(X, +(2,3))$	$X \text{ is } 2+3$
$+(8, -(2))$	$8 + -2$
$>(4,1)$	$4 > 1$
$==(a,a)$	$a == a$
$=(X,a)$	$X = a$

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).
 $x \text{ } Op \text{ } y$
- **Präfix-Operatoren** wie `-`, `+` usw. sind einstellig und werden vor das Argument geschrieben (z.B. `-3`).
 $Op \text{ } x$
- **Postfix-Operatoren** sind einstellig und werden hinter das Argument geschrieben.
 $x \text{ } Op$



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 rechtes 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 linkes 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** $\in \{1, \dots, 1200\}$
- **Typ (+Assoz)** $\in \begin{cases} \{x\bar{f}x, x\bar{f}y, y\bar{f}x\} \text{ wenn } f \text{ Infix ist} \\ \{f\bar{x}, f\bar{y}\} \text{ wenn } f \text{ Präfix ist} \\ \{\bar{x}f, \bar{y}f\} \text{ wenn } f \text{ Postfix ist} \end{cases}$
 - 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.

$y\bar{f}x$: links-assoziativ, $x\bar{f}y$: rechts-assoziativ, $x\bar{f}x$ nicht-assoziativ.



DEFINITION BESTEHENDER OPERATOREN

```
:- op( 1200, xfx, [ :-, --> ]).
:- op( 1200, fx, [ ?- ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1000, xfy, [ ',', ]).
:- op( 700, xfx, [ =, is, =.., ==, \==, =:=, =\=, <, >, =<, >= ]).
:- op( 500, yfx, [ +, - ]).
:- op( 200, fx, [ +, - ]).
:- op( 400, yfx, [*,/,mod]).
```

	3	+	4	+	5
3 + 4 + 5:	0	500	0		
	0	500	500		

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



BEISPIEL: ZWEI VERSCHIEDENE OPERATOREN

$3 + 4 * 5:$

3	+	4	*	5
0	500	0		
500		400	0	

3	+	4	*	5
0	400	0	500	0
0	400	400	500	

$3 + 4 * 5:$

$\ast($	$+$ (3,	4),	5))
500	0	0		
400	500			0

$+($	3,	$\ast($	4	,5))
		400	0	0
500	0	400		



BEISPIEL

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

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



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,5,1].  
true.  
?- k in [3,7,w,5,1].  
false.
```

