

Prolog

6. Kapitel: Listenprädikate

Dozentin: Wiebke Petersen

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

Basisprädikate zur Listenmanipulation

Vier Prädikate zur rekursiven Listenverarbeitung demonstrieren Basistechniken für beliebige 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)`

Zusammenfassung Kapitel 5

- Wir haben gesehen, wie wir mit Prolog rechnen können.
- Wir haben arithmetische Vergleichsoperatoren kennengelernt.
- Wir haben gelernt, wie Akkumulatoren in der rekursiven Listenverarbeitung eingesetzt werden können, um effizienter Prädikate zu erhalten.
- **Keywords:** Rechnen in Prolog mit dem Evaluationsoperator `is`, arithmetische Vergleichsoperatoren, Akkumulatoren.
- **Wichtig:** Die rekursive Verarbeitung von Listen mit Akkumulatoren ist eine zentrale Programmieretechnik in Prolog.
- **Vorsicht:** Die arithmetischen Vergleichsoperatoren und der Operator `is` fordern zwingend sofort evaluierbare Terme. Uninstantiierte Terme führen zu einem Abbruch mit Fehlermeldung.
- **Ausblick Kapitel 6:** Weitere Listenprädikate

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).

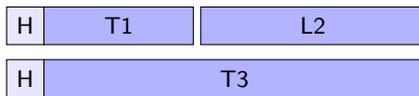
append([H|T1],L2,[H|T3]) :-
    append(T1,L2,T3).
```

```
?- 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([],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.
- Die Konkatenation einer Liste L an eine nichtleere Liste L1 ist L3, wobei der Kopf von L3 identisch mit dem Kopf von L1 ist und der Rest von L3 die Konkatenation von L an den Rest von L1 repräsentiert.



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

```
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] ;
```

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 **konkatenerieren**: `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]).
```

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 Listenabarbeitung 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.

► Übung

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).
```



► Übung

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] ;
X = [a, b, 1, c] ;
X = [a, b, c, 1] .
```

► Übung: deleteall/3

► Übung: permute/2

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)`.

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])
...
```

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)
```

- Wir haben die zentralen Listenprädikate `append/3`, `delete/3` und `reverse/2` kennengelernt.
- Wir haben gelernt, Akkumulatoren für Listenelemente zu verwenden, um effizienter Prädikate zu definieren.
- Wir haben Differenzlisten eingeführt, die die Konkatenation von Listen in einem Schritt ermöglichen.
- Keywords:** `append/3`, `delete/3`, `reverse/2`, Akkumulatorlisten, Differenzlisten.
- Wichtig:** Rekursive Prädikatsaufrufe sind ineffizient und sollten auf ein Minimum beschränkt werden (vgl. `naiverev/2` mit `reverse/2` und `append/3` mit `append_dl/3`).
- Ausblick Kapitel 7:** Definite Clause Grammars (DCG's)

```
% Grammatikregeln:
s(L3):-
    np(L1),
    vp(L2),
    append(L1,L2,L3).
np(L3):-
    det(L1),
    n(L2),
    append(L1,L2,L3).
vp(L3):-
    v(L1),
    np(L2),
    append(L1,L2,L3).
```

```
% Lexikon:
det([eine]).
det([die]).
det([keine]).
n([maus]).
n([katze]).
v([jagt]).
v([klaut]).
```

- Wieviele Sätze können mit dieser Grammatik generiert werden?
- Betrachten Sie die folgende Anfrage im Tracemodus:

```
?- s([keine, katze, klaut, eine, maus]).
```
- Fällt Ihnen eine Möglichkeit ein, die Prädikate für die Grammatikregeln effizienter zu definieren?

- Überlegen Sie sich eine alternative Definition für `sublist/2`.
- Schreiben Sie ein Prädikat `swapfl/2`, das zwei Listen akzeptiert, wenn die eine aus der anderen durch Vertauschen des ersten und des letzten Elements hervorgeht:

```
?- swapfl([a,b,c,d],[d,b,c,a]).
true.
```

Lässt sich ihr Prädikat auch mit Variablen als Argumente in der Anfrage verwenden? Wenn nein, warum nicht?

▸ zurück

Schreiben Sie ein Prädikat `delete_sublist/3`, das drei Listen als Argumente nimmt und gelingt, wenn die dritte Liste das Resultat ist, das man erhält, wenn man alle Vorkommen von Elementen der ersten Liste aus der zweiten Liste löscht.

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

Schreiben Sie ein Prädikat `deleteall/3`, das alle Vorkommen eines Elements aus einer Liste löscht:

```
?- deleteall(b,[a,b,c,d],[a,c,d]).
true.
?- deleteall(1,[1,2,1,2,3,1,2,3,4],[2,2,3,2,3,4]).
true.
?- deleteall(1,[a,b,c],[a,b,c]).
true.
?- deleteall(1,[1,1,1,1],[]).
true.
```

Wie verhält sich ihr Prädikat, wenn sie es mit Variablen an den verschiedenen Argumentpositionen aufrufen?

▸ zurück

Schreiben Sie ein Prädikat `permute/2`, das zwei Listen akzeptiert, wenn die eine Liste aus der anderen durch Permutation der Elemente hervorgeht:

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

▸ zurück

- Schreiben Sie ein Prädikat `palindrom/2`, das Palindrome erkennt. Ein Wort ist ein Palindrom, wenn es von vorne und von hinten gelesen gleich ist.

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

- Schreiben Sie ein Prädikat `wiederhol/2`, das Wiederholwörter erkennt. Ein Wort ist ein Wiederholwort, wenn es aus genau zwei gleichen Teilen besteht.

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

▶ zurück

Schreiben Sie ein Prädikat `flatten/2`, das als Argumente zwei Listen nimmt und gelingt, wenn die zweite Liste die „Verflachung“ der ersten Liste ist:

```
?- flatten([a,b,[c,d]],[a,b,c,d]).
true.
?- flatten([[a,[b,[c,d]]],[a,b,c,d]).
true.
```

- Auf der letzten Folie von Foliensatz 4 finden Sie eine Grammatik, mit der Sie Ableitungsbäume auf ihre Wohlgeformtheit testen können.
- Hier im Kapitel 6 haben wir gesehen, wie wir das Prädikat `append/3` in einer Grammatik zur Generierung von Sätzen einsetzen können.
- Verbinden Sie beide Ansätze zu einem Parser: Ziel ist ein zweistelliges Prädikat `s/2`, das gelingt, wenn das erste Argument ein grammatischer Satz in Form einer Wortliste ist und das zweite der Ableitungsbaum dieses Satzes:

```
?- s([eine, maus, jagt, viele, katzen],
    s(np(d(eine), n(maus)), vp(v(jagt), np(d(viele), n(katzen)))).
true.
```

Gegeben folgende Situation:

- In einer Straße stehen drei Häuser mit den Hausnummern 1, 2 und 3.
- Es gibt ein rotes, ein blaues und ein grünes Haus.
- In der Straße wohnt ein Engländer, eine Japanerin und eine Spanierin.
- In jedem Hau gibt es ein Haustier: eine Schnecke, ein Jaguar und einen Wolf.
- In dem roten Haus wohnt ein Engländer.
- Die Spanierin hält einen Jaguar.
- Die Japanerin wohnt rechts neben dem Haus, in dem die Schnecke gehalten wird.
- Die Schnecke lebt in dem Haus links vom blauen Haus.

Modellieren Sie diese Situation in Prolog und schreiben Sie ein Prädikat, mit dem sie herausfinden können, wer in welchem Haus, welcher Farbe mit welchem Tier lebt. Gibt es mehr als eine Lösung? Denken Sie sich eine zusätzliche Bedingung aus, so dass die Lösung eindeutig wird.

Tipp: Mithilfe von `permute/2` können Sie sich potentielle Hausbelegungen erzeugen, die Sie dann daraufhin überprüfen können, ob sie die angegebenen Bedingungen erfüllen.

