

**Aufgabe 1:** *Listen in Prolog*

---

a) Den Fall der leeren Liste müssen wir hier nicht betrachten, denn eine leere Liste besitzt kein Maximum. Also ist Standardantwort `no` von Prolog für diesen Fall ausreichend. Falls die Liste aus nur einem Element besteht, dann ist insbesondere dieses Element auch das letzte Element der Liste. Falls eine Liste mehr aus mindestens zwei Elementen besteht, also die Form `[_ | Tail]` hat, so berechne das letzte Element der Liste mit einem rekursiven Aufruf von `tail`.

```
last(Item, [Item]).  
last(Item, [_|T]) :- last(Item, T).
```

Die Definition mit `concat` ist noch kürzer, die Liste lässt sich nämlich aus einer beliebigen Liste und einer Liste mit dem letzten Element zusammensetzen:

```
clast(X,L) :- concat(_, [X], L).
```

**Wir wollen diese Definition einmal genauer an einem Beispiel nachvollziehen:** Betrachten wir den folgenden Aufruf und seine folgenden Rekursionsaufrufe:

```
clast(X, [1,2,3]) :- concat (_, [X], [1,2,3]).  
concat(_, [X], [1|[2,3]]) :- concat(_, [X], [2,3]).  
concat(_, [X], [2|[3]]) :- concat(_, [X], [3]).  
concat(_, [X], [3]).  
X= 3  
yes
```

Wir rufen `clast` mit der Liste `[1,2,3]` auf und damit auch `concat` mit dieser Liste. Der Head der Liste wird mit jedem Rekursionsschritt immer weggeworfen, bis wir `concat` mit einer einelementigen Liste für `L3` aufrufen, also das Faktum erreichen. Die Liste wird also rekursiv abgebaut. Da wir nicht konkatenieren ist die erste Liste egal, diese Variable wird nie gelesen, beispielsweise würde bei einem Aufruf `concat(A,[X],[2,3])` die Variable `A` nie unifiziert werden (sie wird weder gelesen noch beschrieben).

b) Falls eine Liste mindestens aus zwei Elementen besteht, so vergleichen wir die ersten beiden Elemente und suchen dann rekursiv in der Restliste, der wir das grössere der zuvor verglichenen Elemente voranstellen, nach dem Maximum der Gesamtliste.

```
maxList([X], X).  
maxList([X,Y|T], MaxItem) :- (X > Y), maxList([X|T], MaxItem).  
maxList([_,Y|T], MaxItem) :- maxList([Y|T], MaxItem).
```

c) Anstatt eines paarweisen Vergleichs haben wir hier eine Addition mit jedem Rekursionsaufruf auszuführen. Damit Prolog die Addition auswertet, werden die beiden Werte einer Hilfsvariable zugewiesen. Diese Hilfsvariable wird dann (analog zur vorherigen Aufgabe) für den Rekursionsaufruf wieder der Restliste vorangestellt.

```
sumList([X], X).  
sumList([X,Y|T], SumList) :- Z is X+Y, sumList([Z|T], SumList).
```

d) Weil die Liste hier aufsteigend oder absteigend sortiert sein darf, brauchen wir hier zwei getrennte Aufrufe, die dann rekursiv die Liste nach aufsteigender bzw. absteigender Sortierung prüfen.

Für Listen mit mindestens zwei Elementen vergleichen wir jeweils die ersten beiden Elemente X und Y und prüfen dann rekursiv, ob die Ordnung auch für Y und die Restliste erfüllt ist.

```
ordered(List):- isAscSorted(List); isDescSorted(List).
```

```
isAscSorted([]).
isAscSorted([_]).
isAscSorted([X,Y|T]):- (X<Y), isAscSorted([Y|T]).
```

```
isDescSorted([]).
isDescSorted([_]).
isDescSorted([X,Y|T]):- (X>Y), isDescSorted([Y|T]).
```

## Aufgabe 2: Wege in Graphen

Der Graph wird durch seine Kanten abgebildet. Da der Graph ungerichtet ist, muss auch die Kante in die Gegenrichtung angegeben werden:

```
weg(a, b).
weg(a, c).
weg(b, a).
weg(b, d).
% [...]
```

a) Zunächst definieren wir eine Regel die eine Pfad von X nach Y sucht und dabei schon besuchte Knoten ignoriert. Falls es einen direkten Weg von X nach Y gibt und Y noch nicht besucht wurde wird die Kante XY zurückgegeben. Im 2. Fall suchen wir einen direkten Nachbarknoten N von X, der noch nicht besucht wurde und suchen anschließend einen Weg von N nach Y, wobei N in die Liste der besuchten Knoten aufgenommen wird:

```
findeWeg(X, Y, Visited, [X,Y]) :-
    weg(X, Y),
    not(member(Y, Visited)).
findeWeg(X, Y, Visited, [X|NY]) :-
    weg(X, Neighbour),
    not(member(Neighbour, Visited)),
    findeWeg(Neighbour, Y, [Neighbour|Visited], NY).
```

Um alle Wege von X nach Y zu finden benutzen wir die Regel findall und findeWeg um eine Liste mit allen Wegen von X nach Y zu erzeugen:

```
alleWege(X, Y, L) :- findall(Path, findeWeg(X, Y, [X], Path), L).
```

b) Um den längsten Weg von X nach Y zu finden definieren wir eine Hilfsregel, die aus einer Liste von Listen die längste Liste zurückgibt.

Wenn die Liste nur eine interne Liste enthält, wird diese Liste zurückgegeben. Im anderen Fall wird die längste Liste aus der Restliste und mit der ersten Liste verglichen und die längere der beiden ist das Ergebnis des Aufrufs.

```
longestList([[Head|Tail]], [Head|Tail]) :- !.
longestList([Head|Tail], MaxTailList) :-
    longestList(Tail, MaxTailList),
    length(Head, HeadLength),
```

```

    length(MaxTailList, MaxTailListLength),
    HeadLength < MaxTailListLength,
    !.
longestList([MaxList|_], MaxList) :- !.

```

Um den längsten Pfad zwischen X und Y zu finden, werden alle möglichen Pfade berechnet und daraus der längste ausgewählt.

```

maxWeg(X, Y, MaxWeg) :-
    alleWege(X, Y, Ws),
    longestList(Ws, MaxWeg).

```

---

## vollständige Quelltexte

---

### Quelltext zu Aufgabe 1:

```

% Last(Item, List)
% liefert das letzte Element der Liste List, indem
% wir solange die Funktion rekursiv aufrufen und
% den Kopf der Liste wegwerfen, bis List nur noch
% aus einem Element besteht und zwar dem letzten.
% Aufruf: last(+,+).
last(Item, [Item]).
last(Item, [_|T]):- last(Item, T).

% concat, aus der Vorlesung bekannt:
concat([], L,L).
concat([X|L1], L2, [X|L3]):- concat(L1,L2,L3).

% last (mit Hilfe von concat),
% der Aufruf von clast ist analog zu last
% allerdings wird concat solange rekursiv aufgerufen,
% bis L nur noch die einelementige Liste mit dem
% letzten Element ist.
% Aufruf: clast(+,+).
clast(X,L):- concat(_, [X],L).

% maxList(List, MaxItem)
% liefert das größte Element einer Liste, indem wir
% paarweise jeweils die ersten beiden Elemente
% vergleichen und das grössere in den rekursionsaufruf
% übergeben. Am Schluss besteht die Liste nur noch aus
% dem größten Element.
% Aufruf: maxList(+,+).
maxList([X], X).
maxList([X,Y|T], MaxItem):- (X > Y),maxList([X|T], MaxItem).
maxList([X,Y|T], MaxItem):- (X < Y), maxList([Y|T], MaxItem).

% sumList([x],X).
% Wir verfahren hier analog zur vorherigen Aufgabe, bilden
% allerdings rekursiv immer die Summe der ersten beiden
% Elemente der Liste.
% Aufruf: sumList(+,+).
sumList([X], X).
sumList([X,Y|T], SumList):- Z is X+Y, sumList([Z|T], SumList).

```

```

% ordered(List)
% prüft ob eine Liste aufsteigend oder absteigend sortiert
% ist. Dafür werden zwei getrennte Aufrufe verwendet, die mit
% einem logischen ODER verknüpft sind. Auch hier wird wieder
% paarweise geprüft.
% Aufruf: ordered(+).
ordered(List):- isAscSorted(List); isDescSorted(List).

isAscSorted([]).
isAscSorted([_]).
isAscSorted([X,Y|T]):- (X<Y) ,isAscSorted([Y|T]).

isDescSorted([]).
isDescSorted([_]).
isDescSorted([X,Y|T]):- (X>Y) ,isDescSorted([Y|T]).

```

### Quelltext zu Aufgabe 2:

```

% weg(X, Y): Es gibt einen Weg von X nach Y
% alle möglichen Wege
weg(a, b).
weg(a, c).
weg(b, a).
weg(b, d).
weg(b, e).
weg(c, a).
weg(c, d).
weg(c, e).
weg(d, b).
weg(d, c).
weg(d, e).
weg(e, b).
weg(e, c).
weg(e, d).

% findeWeg(X, Y, Visited, Path): findet einen Weg von X nach Y der die
% Knoten in
% der Liste Visited nicht benutzt und gibt das Ergebnis in Path zurück.
% Wenn es einen direkten Weg von X nach Y gibt und Y nicht in Visited
% enthalten
% ist, dann ist ein Ergebnis der direkte Weg von X nach Y
findeWeg(X, Y, Visited, [X,Y]) :-
    weg(X, Y),
    not(member(Y, Visited)).

% Ein Weg von X nach Y geht über einen direkten Nachbarn von X der noch
% nicht in
% der Liste Visited enthalten ist und dann von diesem Nachbarn nach Y,
% wobei der
% Nachbar in der Liste Visited jetzt enthalten ist.
findeWeg(X, Y, Visited, [X|NY]) :-
    weg(X, Neighbour),
    not(member(Neighbour, Visited)),
    findeWeg(Neighbour, Y, [Neighbour|Visited], NY).

% alleWege(X, Y, L): findet alle Wege von X nach Y und gibt das Ergebnis
% in L

```

```

% zurück.
alleWege(X, Y, L) :- findall(Path, findeWeg(X, Y, [X], Path), L).

% printWege(X, Y): Gibt alle Wege von X nach Y aus.
printWege(X, Y) :-
    alleWege(X, Y, Wege),
    length(Wege, AnzahlWege),
    AnzahlWege > 0,
    sort(Wege, SWege),
    printList(SWege).

% printList(List): Gibt die Element von List aus
% Bei einer leeren Liste ist nichts zu machen.
printList([]).

% Ansonsten das erste Element ausgeben und anschließend den Rest der Liste
printList([Head|Tail]) :-
    print(Head),
    nl,
    printList(Tail).

% maxWeg(X, Y, MaxWeg): Gibt den längsten Weg von X nach Y in W zurück
maxWeg(X, Y, MaxWeg) :-
    alleWege(X, Y, Ws),
    longestList(Ws, MaxWeg).

% longestList(Lists, MaxList): Gibt in W die längste Liste aus Ws zurück
longestList([[Head|Tail]], [Head|Tail]) :- !.
longestList([Head|Tail], MaxTailList) :-
    longestList(Tail, MaxTailList),
    length(Head, HeadLength),
    length(MaxTailList, MaxTailListLength),
    HeadLength < MaxTailListLength,
    !.
longestList([MaxList|_], MaxList) :- !.

```

---

## Testläufe

---

### Testläufe zu Aufgabe 1:

Das letzte Element der leeren Liste ausgeben:

```
1 ?- last(X, []).
```

No

Das letzte Element einer beliebigen Liste:

```
2 ?- last(X, [1,2,34]).
```

```
X = 34 ;
```

No

```
3 ?- last(X, [1,2,34, [2],6,8,[4]]).
```

```
X = [4] ;
```

No

Analoge Ergebnisse liefert auch `clast`, also die Implementierung von `last` mit Hilfe von `concat`:

```
6 ?- clast(X, []).
```

No

```
7 ?- clast(X, [2,5,[9,9], 7]).
```

```
X = 7 ;
```

No

Das Maximum einer Liste existiert nur für nichtleere Listen.

```
10 ?- maxList([], X).
```

No

```
11 ?- maxList([1,2,5], X).
```

```
X = 5
```

Wir können sogar die Summe von Elementen aus Listen einer Liste bilden:

```
13 ?- sumList([1,2,3,4,5], X).
```

```
X = 15 ;
```

No

```
15 ?- sumList([[1],[2],[6]], X).
```

```
X = 9 ;
```

No

Und einige Testfälle für die Sortierung von Listen:

```
16 ?- ordered([]).
```

Yes

```
17 ?- ordered([1,2,3,4]).
```

Yes

```
18 ?- ordered([4,3,2,1]).
```

Yes

```
19 ?- ordered([4,3,7,2,1]).
```

No

### Testläufe zu Aufgabe 2:

```
3 ?- alleWege(a, b, AB).
```

```
AB = [[a, b], [a, c, d, b], [a, c, d, e, b], [a, c, e, b], [a, c, e, d  
|...]] ;
```

No

```
4 ?- alleWege(c, d, AB).
```

```
AB = [[c, d], [c, a, b, d], [c, a, b, e, d], [c, e, d], [c, e, b, d]] ;
```

No

```
5 ?- printWege(a, b).  
[a, b]  
[a, c, d, b]  
[a, c, d, e, b]  
[a, c, e, b]  
[a, c, e, d, b]
```

Yes

```
6 ?- printWege(a, e).  
[a, b, d, c, e]  
[a, b, d, e]  
[a, b, e]  
[a, c, d, b, e]  
[a, c, d, e]  
[a, c, e]
```

Yes

```
7 ?- printWege(b, a).  
[b, a]  
[b, d, c, a]  
[b, d, e, c, a]  
[b, e, c, a]  
[b, e, d, c, a]
```

Yes

```
8 ?- printWege(e, a).  
[e, b, a]  
[e, b, d, c, a]  
[e, c, a]  
[e, c, d, b, a]  
[e, d, b, a]  
[e, d, c, a]
```

Yes

```
9 ?- maxWeg(a, b, MaxWeg).
```

```
MaxWeg = [a, c, d, e, b] ;
```

No

```
10 ?- maxWeg(b, a, MaxWeg).
```

```
MaxWeg = [b, d, e, c, a] ;
```

No

```
11 ?- maxWeg(a, e, MaxWeg).
```

```
MaxWeg = [a, b, d, c, e] ;
```

No

```
12 ?- maxWeg(e, a, MaxWeg).
```

```
MaxWeg = [e, b, d, c, a] ;
```

No