Lösung

Aufgabe 13

13.1) Gegeben seien zwei Listen, L1 und L2. Schreiben Sie ein Prädikat `durchschnitt/3`, das eine Liste erzeugt, in der nur Elemente vorkommen, die sowohl in L1 als auch in L2 vorkommen.

L1=[1,2,3,4],L2=[3,4,5,6],L3=[3,4]

```
% Ohne Cut
durchschnitt([],_,[]).
durchschnitt([X|R],L,Erg):-not(member(X,L)),durchschnitt(R,L,Erg). %Kopf weggeworfen
durchschnitt([X|R],L,[X|L2]):-member(X,L),durchschnitt(R,L,L2). %Kopf wird auf das Ergebnis geschrieben
                                                                    %Rekursiv erfolgt der Zusammenbau

% mit Cut und Elimination Doppelter
durchschnitt([],_,[]).
durchschnitt([X|R],L,Erg):-not(member(X,L)),!,durchschnitt(R,L,Erg). %Cut damit zweite Alternative nicht durch
                                                                    % Backtracking erreichbar ist

durchschnitt([X|R],L,[X|L2]):-loesche_all(X,L,Zwisch),
                               durchschnitt(R,Zwisch,L2).

%%Hilfsprädikat
loesche_all(_,[],[]).
loesche_all(X,[X|R],L):-loesche_all(X,R,L).
loesche_all(X,[Y|R],[Y|L]):-X\=Y,loesche_all(X,R,L).

%?- durchschnitt([2,2,3,4],[2],Ergebnis).
%Ergebnis = [2] ; mehrfach auftretende Zahlen werden eliminiert.
%false.
```

13.2) Implementieren Sie nun das Prädikat `differenz/3` indem nur Elemente vorkommen die exklusiv in L1 vorkommen.

L1=[1,2,3,4],L2=[3,4,5,6],L3=[1,2]

```
% Ohne Cut
differenz([],_,[]).
differenz([X|R],L,[X|Erg]):-not(member(X,L)),differenz(R,L,Erg).
differenz([X|R],L,Erg):-member(X,L),differenz(R,L,Erg).
% Mit Cut
differenz([],_,[]).
differenz([X|R],L,[X|Erg]):-not(member(X,L)),!,differenz(R,L,Erg).
differenz([_R],L,Erg):-differenz(R,L,Erg).
%?- differenz([1,2,3,4],[2,4,5],Erg).
%Erg = [1, 3].
%
```

13.3) Implementieren Sie ein Prädikat `vereinigungsmenge/3`, das die Vereinigungsmenge von zwei Listen erstellt. Sie dürfen annehmen, dass die Eingabelisten Mengen sind.

```
%Lösung 1
vereinigung(L1,L2,Erg):-differenz(L1,L2,DifferenzMenge),
    append(DifferenzMenge,L2,Erg).

%Lösung 2
%ohne Cut
vereinigung([],L,L).
vereinigung([X|R],L1,Erg):-member(X,L1),vereinigung(R,L1,Erg).
vereinigung([X|R],L1,[X|Erg]):-not(member(X,L1)),vereinigung(R,L1,Erg).
%mit Cut
vereinigung([],L,L).
vereinigung([X|R],L1,Erg):-member(X,L1),!,vereinigung(R,L1,Erg).
vereinigung([X|R],L1,[X|Erg]):-vereinigung(R,L1,Erg).

%?- vereinigung([1,2,3,4],[2,4,5],Erg).
%Erg = [1, 3, 2, 4, 5].
%
```

13.4) Implementieren Sie nun das Prädikat `durchschnittsmenge/3`, das die Durchschnittsmenge von zwei Listen erstellt. Wiederum dürfen Sie annehmen, dass die Eingabelisten Mengen sind. Überlegen Sie ob es einen Unterschied zu Aufgabe 13.1 gibt.

```
durchschnitt_menge([],_,[]).
durchschnitt_menge([X|R],L,Erg):-not(member(X,L)),!,durchschnitt_menge(R,L,Erg).
durchschnitt_menge([X|R],L,[X|L2]):-durchschnitt_menge(R,L,L2).

%?- durchschnitt_menge([2,3,4],[2],Er).
%Er = [2].

%?- durchschnitt_menge([2,2,3,4],[2],Er).
%Er = [2, 2].
% mehrfach auftretende Zahlen (=keine Menge als Eingabeliste) werden nicht reduziert.
%
```

14.1) Gegeben sei ein kreisfreier Graph mit den Knoten 1 -7. Die Kanten des Graphen sollen

```
kante(1,2).
kante(1,3)
```

```
kante(2,4)
kante(2,5)
kante(3,4)
kante(4,7)
kante(5,6)
kante(6,7)
```

Schreiben Sie ein Prädikat `pfad`, das alle Pfade im Graphen berechnet und als Listen ausgibt: Beispielausgabe $X = [1,2,4,7]$. Mit ausgelöstem Backtracking sollen alle Pfade im Graphen ausgegeben werden können. Es hilft, wenn Sie den Graphen aufzeichnen. Zusätzlich zum rekursiven Pfadprädikat müssen Sie eine Liste mit den Knoten des Pfads erzeugen.

```
%Lösung 1a
pfad(X,Y,[X,Y]):-kante(X,Y).
pfad(X,Y,[X|L1]):-kante(X,Z),pfad(Z,Y,L1).
```

```
%- pfad(1,7,Pfad).
%Pfad = [1, 2, 4, 7] ;
%Pfad = [1, 2, 5, 6, 7] ;
%Pfad = [1, 3, 4, 7] ;
```

```
%Lösung 1b
pfad2([X,Y]):-kante(X,Y).
pfad2([X|[Y|Z]):-kante(X,Y),pfad2([Y|Z]).
```

```
%Lösung 2
%Mit Akkumulator
pfad_acc(X,Y,Pfad):-pfad_acc(X,Y,[],PfadR), reverse(PfadR,Pfad).
pfad_acc(X,Y,Acc,[Y,X|Acc]):-kante(X,Y).
pfad_acc(X,Y,Acc,Pfad):-kante(X,Z),pfad_acc(Z,Y,[X|Acc],Pfad).
```

14.2) Benutzen Sie eine Schleifenkonstruktion wie in der Vorlesung vorgestellt um alle Wege auszugeben. Hinweis, in `drucke(X)` hatten wir schon das `write` Prädikat benutzt.

```
weg(X,Y):-kante(X,Y).
weg(X,Y):-kante(X,Z),weg(Z,Y).
zeigeWeg:-
    weg(X,Y), %sucht den Weg zwischen zwei Punkten
    write('Es gibt den Weg('),
    write(X), % gibt Anfangspunkt aus
    write(','),
    write(Y), % gibt Endpunkt aus
    write(').\n'),
    %nl,
    fail.
```

```

% Prolog sucht zuerst alle Wege, bis es keine mehr findet. Dann scheitert die erste Klausel. zeigeWeg
% Nach dem Scheitern der ersten Klausel wird zweite Klausel versucht. Diese ist einfach ein Fakt.
%%Jetzt endet das Programm mit true.
%%%%%%%%
?-zeigeWeg.

```

```

Es gibt den Weg(1,2).
Es gibt den Weg(1,3).
Es gibt den Weg(2,4).
Es gibt den Weg(2,5).
Es gibt den Weg(3,4).
Es gibt den Weg(4,7).
Es gibt den Weg(5,6).
Es gibt den Weg(6,7).
Es gibt den Weg(1,4).
Es gibt den Weg(1,5).
Es gibt den Weg(1,7).
Es gibt den Weg(1,6).
Es gibt den Weg(1,7).
Es gibt den Weg(1,4).
Es gibt den Weg(1,7).
Es gibt den Weg(2,7).
Es gibt den Weg(2,6).
Es gibt den Weg(2,7).
Es gibt den Weg(3,7).
Es gibt den Weg(5,7).
true.

```

14.3) Fügen Sie die folgenden Kanten ein:

```

kante(6,1).
kante(7,2).
kante(7,3).

```

Der Graph ist nun zyklisch. Schreiben Sie ein Programm, das alle zyklischen Pfade des Graphen in der Form von Listen berechnet. Hinweis: Die Listen dürfen keine doppelten Elemente enthalten.

Loesung:

```

% zyklischer Kreis: 4->7->2->4->7->2->4....
zyklenfrei(X,Y,P):-zyklenfrei(X,Y,[],PR), reverse(PR,P).% Alle besuchten Punkte werden in einem Akkumulator
gesammelt.
zyklenfrei(X,Y,Akk,[Y,X|Akk]):-kante(X,Y),not(member(Y,Akk)).
% Einen zyklischen Weg gibt es von X nach Y, genau dann wenn kein Element in der Akkumulatorliste zweimal
auftritt.
zyklenfrei(X,Y,Akk,Pfad):-kante(X,Z),not(member(Z,Akk)),zyklenfrei(Z,Y,[X|Akk],Pfad).% Sammlung der neuen
Punkte.

```

Alternative:

```

pfad2zyklenfrei([X,Y]):-kante(X,Y).
pfad2zyklenfrei([X|[Y|Z]]):-kante(X,Y),pfad2zyklenfrei([Y|Z]), not(member(X,[Y|Z])).

```

15.1) Betrachten Sie folgendes Prolog Programm:

```
p(1).
p(2):-!.
p(3).
```

Geben Sie die Antworten folgender Anfragen an:

- (a) ?- p(X).
- (b) ?- p(X),p(Y).
- (c) ?- p(X),!,p(Y).

```
%15.1)
p(1).
p(2):-!.
p(3).
% Mit Cut wird die Suche nach weiteren Lösungen verhindert. Im Programm
% wird die dritte Klausel nie aufgerufen.
%?- p(X). X = 1 ; X = 2.
%?- p(X),p(Y).
```

erste Ausgabe:

```
%X = 1,      geht topdown mit goal 1 in das Programm und erreicht als erstes p(1)
%X = 1,      X wird mit 1 unifiziert, dann wird das zweite goal p(Y) aufgerufen
%Y = 1 ;     wieder topdown in das Programm, p(1) wird aufgerufen, Y mit 1 unifiziert.
              wir haben marker auf p(1) für goal p(X) und p(19 für goal p(Y)
```

Backtracking: es wird versucht das goal am weitesten rechts anders zu lösen, also p(Y).
Alles andere bleibt gleich, insbesondere bleibt p(X) auf p(1) stehen.
p(Y) geht in das Programm, findet einen Backtracking-Marker auf p(1) und nimmt deshalb die
nächstmögliche Lösung p(2). Y ist also 2.

```
%X = 1,
%X = 2 ;
```

Backtracking: es wird versucht das goal am weitesten rechts anders zu lösen, also p(Y).
Da sowohl p(1) und p(2) schon Marker haben für p(Y) und es keine weiteren Alternativen gibt
Muss in der goalliste nach links nach weiteren Lösungen gesucht werden.
Also wird versucht p(X) neu zu belegen. Es ergibt sich p(2) als Lösung.
Von hier aus wird die Goalliste nach rechts neu bewiesen.

Es muss also p(Y) neu bewiesen werden.

In allen Varianten.

Backtracking löscht alle Belegungen rechts vom Goal das unter Backtracking neu belegt werden soll.

Die erste Möglichkeit $p(Y)$ neu zu beweisen ist $p(1)$.

```
%X = 2,  
%Y = 1 ;
```

Backtracking: Die zweite Möglichkeit $p(Y)$ unter der Belegung $p(X) = p(2)$ ist $p(2)$.
Daher die Lösungsreihenfolge.

```
%X = 2,  
%Y = 2.
```

```
%?- p(X),!,p(Y).  
%X = 1,  
%Y = 1 ;  
%X = 1,  
%Y = 2.
```

%Der Cut schneidet für $p(X)$ die Alternativen ab. Nicht jedoch für $p(Y)$.

15.2)

Definieren Sie eine Relation, die eine EingabeZahl in drei mögliche Klassen einteilt: Positive ($X > 0$), Null ($X == 0$) und Negative ($X < 0$). Benutzen Sie den cut, um das Programm möglichst effizient zu machen.

```
%15.2)  
zahl(X):-X>0,!,  
    write(X),  
    write(' ist ein positive Zahl.').  
zahl(X):-X==0,!,  
    write('Sie haben Null eingegeben.').  
zahl(X):- write(X),  
    write(' ist ein negative Zahl.').
```