

SS2010
BAI2-LBP Gruppe 1 Team 07
Lösung zur Aufgabe 1

R. C. Ladiges, D. Fast

13. April 2010

Inhaltsverzeichnis

1 Aufgabe 1	3
1.1 writelnListe	3
1.1.1 Aufgabenstellung	3
1.1.2 Entwurf	3
1.1.3 Quelltext	3
1.2 touchListe	3
1.2.1 Aufgabenstellung	3
1.2.2 Entwurf	4
1.2.3 Quelltext	4
1.2.4 Anmerkung	4
1.3 substitute	4
1.3.1 Aufgabenstellung	4
1.3.2 Entwurf	4
1.3.3 Quelltext	5
1.4 entfDuplikate	5
1.4.1 Aufgabenstellung	5
1.4.2 Entwurf	5
1.4.3 Quelltext	6
1.4.4 Anmerkung	6
1.5 diffL	6
1.5.1 Aufgabenstellung	6
1.5.2 Entwurf	6
1.5.3 Quelltext	7
1.5.4 Anmerkung	7
1.6 kombiniere	7
1.6.1 Aufgabenstellung	7
1.6.2 Entwurf	8
1.6.3 Quelltext	8
1.7 zyklisch_succ	8
1.7.1 Aufgabenstellung	8
1.7.2 Entwurf	8
1.7.3 Quelltext	9
1.7.4 Anmerkung	9
1.8 Der Nibelungen Not	10
1.8.1 Aufgabenstellung	10
1.8.2 Entwurf	10
1.8.3 Quelltext	10
1.8.4 Anmerkung	11

1 Aufgabe 1

1.1 writelnListe

1.1.1 Aufgabenstellung

Implementieren Sie zwei Prädikate `writelnListe-nr` und `writelnListe-er`, die die Elemente einer Liste einmal per **naiver Rekursion (nr)** und einmal per **Endrekursion (er)** zeilenweise ausgeben:

```
7 ?- writelnListe-er([1,2,3]). 8 ?- writelnListe-nr([1,2,3]).
1                               ende
2                               3
3                               2
ende                             1
yes                              yes
```

1.1.2 Entwurf

Die beiden Prädikate `writelnListe-nr` und `writelnListe-er` werden fast identisch implementiert.

Die Argumentliste wird in einen Kopf und Rest zerteilt. Der Kopf wird ausgegeben und der Rest geht in die Rekursion bis zur Abbruchbedingung (leere Liste als Eingabe), bei welcher `ende` ausgegeben wird.

Der bedeutende Unterschied zwischen den beiden Prädikaten liegt in der Tatsache, dass bei der **Endrekursion** erst der Kopf ausgegeben wird und dann der rekursive Aufruf auf den Rest ausgeführt wird. Bei der **naiven Rekursion** ist dies genau umgekehrt, hier wird erst die Rekursion auf den Rest aufgerufen und dann erst der Kopf ausgegeben.

1.1.3 Quelltext

```
% Naive Rekursion
% writelnListe-nr([1,2,3]). -> ende 3 2 1 yes
writelnListe-nr([]) :- write('ende'). %Rekursionsabbruch
writelnListe-nr([K|R]) :- writelnListe-nr(R), nl, write(K).
% Endrekursion
% writelnListe-er([1,2,3]). -> 1 2 3 ende yes
writelnListe-er([]) :- write('ende'). %Rekursionsabbruch
writelnListe-er([K|R]) :- write(K), nl, writelnListe-er(R).
```

1.2 touchListe

1.2.1 Aufgabenstellung

Implementieren Sie ein Prädikat `touchListe(InList,OutNRLList,OutERList)`, dass die Liste `InList` einmal **zerlegt** (mittels **[.]**-Operator) und diese per **naiver Rekursion** (`OutNRLList`) und **Endrekursion** (`OutERList`) ausschließlich mittels dem **[.]**-Operator wieder **zusammenbaut**.

```
9 ?- touchListe([1,2,3,4],OutNRLList,OutERList).
OutNRLList = [1, 2, 3, 4],
OutERList = [4, 3, 2, 1].
```

1.2.2 Entwurf

Bei der naiven Rekursion entfernen wir von `InList` sowie von `OutNRLList` das gleiche Kopfelement, und geben den Rest von beiden rekursiv an `touchListe` weiter, bis beide Listen leer sind (Rekursionsabbruch).

Bei der Endrekursion übergeben wir die `InList` an ein eigenes Unterprädikat namens `mydrehe`, welches mit Hilfe von `myappend` die Elemente dreht und ausgibt.

Beide Prädikate, `mydrehe` und `myappend`, verwenden nur den **[.]**-Operator und Rekursion, jedoch keine in Prolog vorhandenen Prädikate.

1.2.3 Quelltext

```
% touchListe([1,2,3,4],NRL,ERL). -> NRL=[1,2,3,4], ERL=[4,3,2,1].
touchListe(In,OutNR,OutER):- touchListe(In,OutNR,[],OutER),!.
touchListe([],[],X,X):-!.
touchListe([K|IR],[K|NR],Akku,Out):- touchListe(InR,NR,[K|Akku],Out).
```

1.2.4 Anmerkung

Nach Rücksprache mit unserem Dozenten Prof. Dr. Christoph Klauck haben wir einen anderen Lösungsweg implementiert der um einiges Effektiver ist. Anstatt `InList` mit `mydrehe` und `myappend` zu drehen, um `OutERList` zu erhalten, verwenden wir einen Akkumulator. Wir leiten beim Aufruf von `touchListe\3` die Parameter an `touchListe\4` weiter, und initialisieren den Akkumulator mit der leeren Liste. Jedes mal wenn wir nun einen rekursiven Aufruf tätigen packen wir den Kopf in den Akkumulator. Dadurch gelangen die Elemente in umgekehrter Reihenfolge in den Akkumulator bis zum Rekursionabbruch. Von da an wird die fertig gedrehte Liste über `Out` durch die Rekursionen weitergeleitet bis zur Ausgabe von `touchListe\3`.

1.3 substitute

1.3.1 Aufgabenstellung

Implementieren Sie ein Prädikat `substitute(X,A,InList,OutList)`, dass in der Liste `InList` jedes Vorkommen von **X** durch **A** ersetzt.

```
substitute(a,z,[a,b,c,b,a],X).
X = [z,b,c,b,z]
```

```
substitute(a,z,[b,c,d,c,b],X).
X = [b,c,d,c,b]
```

1.3.2 Entwurf

Zur Implementierung dieses Prädikats ist eine Aufteilung in drei Fälle, die jeweils zutreffen können, möglich:

1. Abbruchbedingung

Wenn `InList` leer ist, kann `OutList` auch nur leer sein. Dabei spielt die Belegung von **X** und **A** keine Rolle.

2. Rekursiver Fall

Wenn **X** mit dem Kopf von `InList` übereinstimmt, bestimmen wir den Kopf von `OutList` als **A**. Den Rest von `InList` geben wir dann in die Rekursion um ihn zu substituieren zum Rest von `OutList`.

3. Rekursiver Fall

Wir schicken, genauso wie beim ersten Fall den Rest der `InList` in die Rekursion um den Rest von `OutList` zu erhalten. Der von **X** verschiedene Kopf von `InList` ist in diesem Fall identisch zum Kopf von `OutList`.

1.3.3 Quelltext

```
% substitute(X,A,InList,OutList) % ersetze jedes X in InList mit A
% substitute(a,z,[a,b,c,b,a],X). -> X=[z,b,c,b,z]
% substitute(a,z,[b,c,d,c,b],X). -> X=[b,c,d,c,b]
substitute(_,_,[],[ ]):-!.
substitute(X,A,[X|IR],[A|OR]):- substitute(X,A,IR,OR),!.
substitute(X,A,[K|IR],[K|OR]):- substitute(X,A,IR,OR),!.
```

1.4 entfDuplikate

1.4.1 Aufgabenstellung

Implementieren Sie ein Prädikat `entfDuplikate(ListeIn,ListeOut)`, das dann zutrifft, wenn `ListeOut` aus `ListeIn` durch Entfernen aller Mehrfachvorkommen von Elementen entsteht! Testen Sie Ihr Prädikat auf geeigneten Daten. Implementieren Sie auch eine Variante, bei der nicht das erste Vorkommen eines Elements in der Ergebnisliste enthalten ist, sondern das letzte (bzw. umgekehrt), d.h. implementieren Sie **zwei Varianten**, bei der einmal das erste Vorkommen behalten wird und einmal das letzte Vorkommen. Als vordefiniertes Prädikat darf nur `member` verwendet werden!

Beispiel: `entfDuplikate([1, 2, 3, 3, 1, 2], Lout)` soll mit der einen Variante als Ergebnis `Lout=[1, 2, 3]` und mit der anderen Variante `L=[3, 1, 2]` erzeugen.

1.4.2 Entwurf

Zur Implementierung der ersten Variante des Prädikats `entfDuplikate` schlagen wir vor, dass Prädikat in drei Fälle aufzuteilen, welche jeweils zutreffen können:

1. Abbruchbedingung

Wenn `ListeIn` leer ist kann sie keine Duplikate haben und folglich ist `ListeOut` auch leer.

2. Rekursiver Fall

Wir unterteilen `ListeIn` in Kopf und Rest und überprüfen dann mit dem `member` Prädikat ob der Kopf im Rest vorkommt. Falls dies der Fall ist ignorieren wir den Kopf und behandeln rekursiv den Rest, dessen Ergebnis `ListeOut` ist.

3. Rekursiver Fall

Wenn der Kopf von `ListeIn` nicht im Rest vorkommt, kombinieren wir ihn, da wir wissen dass es nicht noch einmal im Rest vorhanden ist, mit dem Rest von `ListeOut`, den wir über den rekursiven Aufruf auf den Rest von `ListeIn` erhalten, zu `ListeOut`.

Die Umsetzung der zweiten Variante ist mit Zuhilfenahme eines eigens definierten Unterprädikats namens `entfElem` (siehe 1.5) recht einfach. Sie hat die gleiche Abbruchbedingung wie die erste Variante und nur einen Rekursiven Fall. Es wird von `ListeIn` sowie von `ListeOut` der gleiche Kopf abgesplittet, welchen wir per `entfElem` aus den Rest von `ListeIn` entfernen. Die so erhaltene „gereinigte“ Liste geben wir in die Rekursion rein, und erhalten den Rest von `ListeOut`.

1.4.3 Quelltext

```
% entfDuplikateA([1,2,3,3,1,2],Lout). -> Lout=[3,1,2]
entfDuplikateA([],[]).
entfDuplikateA([K|InR],Out):-member(K,InR),!,entfDuplikateA(InR,Out).
entfDuplikateA([K|InR],[K|OutR]):-entfDuplikateA(InR,OutR).

% entfDuplikateB([1,2,3,3,1,2],Out). -> Out=[1,2,3]
entfDuplikateB(In,Out):-entfDuplikateB(In,[],Out).
entfDuplikateB([],Akku,Akku):-!.
entfDuplikateB([K|R],Ak,Out):-member(K,Ak),!,entfDuplikateB(R,Ak,Out).
entfDuplikateB([K|R],Ak,Out):-append(Ak,[K],Tmp),
    entfDuplikateB(R,Tmp,Out),!.
```

1.4.4 Anmerkung

Wie bei Aufgabe 1.2 haben wir das 2. Prädikat `entfDuplikateB` nach Rücksprache mit unserem Dozenten anders gelöst, da die Lösung über `entfElem` nicht effizient genug ist. Ähnlich wie bei Aufgabe 1.2 bedienen wir uns hier einem Akkumulator. Er wird bei den Rekursionaufrufen mit dem Kopf der `ListeIn` gefüllt, falls er noch nicht im Akkumulator vorhanden ist. Beim Rekursionsabbruch steht in Akkumulator die gewünschte `ListeOut`, die wir bis nach draußen durch die Rekursion „weiterreichen“.

1.5 diffL

1.5.1 Aufgabenstellung

Implementieren Sie ein Prädikat `diffL(Liste1,Liste2,Liste1_diff_Liste2)`, dass die Differenz von zwei Listen erzeugt, d.h. im Resultat `Liste1_diff_Liste2` sind nur die Elemente, die nur in genau einer der beiden Listen vorkommen. Bei Mehrfachvorkommen ist nur entsprechend der vorhandenen Anzahl zu reagieren. Die Reihenfolge spielt keine Rolle (Betrachtung als Menge). Als vordefiniertes Prädikat darf nur `member` verwendet werden!

```
10 ?- diffL([1,1,2,2,3,4,5,6,7,8,9,0],[1,2,2,6,6,10,11,12],Resultat).
    Resultat = [1, 3, 4, 5, 7, 8, 9, 0, 6, 10, 11, 12]
```

1.5.2 Entwurf

Diese Aufgabe kann man dadurch lösen, dass man sich ein Unterprädikat `addtoList(A,B,C)` definiert, welches eine gegebene Liste **A** Element für Element zu einer weiteren Liste **B** hinzufügt oder daraus entfernt und als Liste **C** ausgibt. Das zweimalige Ausführen des Unterprädikates, einmal auf `Liste1` und einmal auf `Liste2`, mit Zuhilfenahme einer `Temp` Variable führt zur Ausgabeliste `Liste1_diff_Liste2`.

Hierbei übernimmt `addtolist` die „Denkarbeit“, in dem es je nach Vorkommen eines Elementes aus **A** in **B** unterschiedlich reagiert:

- Wenn das jeweilige Element aus **A** noch nicht in **B** enthalten ist, wird es angehängt und als **C** ausgegeben.
- Beim gegenteiligen Fall, wenn das Element aus **A** in **B** enthalten ist, wird es aus **B** per Unterprädikat `entfElem` entfernt und als **C** ausgegeben.

Das selbst definierte Prädikat `entfElem` geht rekursiv alle Elemente einer gegebenen Liste durch, und überprüft ob ein gegebenes Element in der Liste vorhanden ist und wenn ja entfernt es dieses.

Hierfür wird lediglich rekursiv geguckt ob das gegebene Element im Kopf der Liste ist und verbindet die Ausgabe entsprechend mit dem Kopf oder nicht.

1.5.3 Quelltext

```
% diffL(Liste1,Liste2,Liste1_diff_Liste2)
%   diffL([1,1,2,2,3,4,5,6,7,8,9,0],[1,2,2,6,6,10,11,12],Resultat).
%   -> Resultat = [1, 3, 4, 5, 7, 8, 9, 0, 6, 10, 11, 12]
diffL([],X,X):-!.
diffL([K|R],L,Out):- member(K,L),!,entfElem(K,L,Temp),diffL(R,Temp,Out).
diffL([K|R],L,[K|OutR]):- diffL(R,L,OutR),!.

entfElem(_,[],[]):-!.
entfElem(E,[E|R],R):-!.
entfElem(E,[K|R],[K|OutR]):- entfElem(E,R,OutR).
```

1.5.4 Anmerkung

In dieser Lösung haben wir die Funktionalität des im Entwurf erwähnten Prädikats `addtolist` leicht verändert in `diffL` verlagert. Wenn der jeweilige Kopf (also eines der Elemente) von `Liste1` in `Liste2` vorhanden ist, entfernen wir ihn aus `Liste2` und gehen damit in die Rekursion. Andersrum, also wenn der Kopf von `Liste1` nicht in `Liste2` ist, lassen wir ihn drin und gehen mit der ganzen `Liste2` in die Rekursion.

1.6 kombiniere

1.6.1 Aufgabenstellung

Implementieren Sie ein Prädikat `kombiniere(InfoListe,Daten,Kombination)`, dass zweier Tupel einer Information aus der `InfoListe` mit den `Daten` erzeugt und alle so erzeugten Kombinationen in der Variablen `Kombination` zurück gibt. In den `Daten` können jedoch die Informationen vorhanden sein; in diesem Fall wird dafür kein zweier Tupel erzeugt!

```
4 ?- kombiniere([a,b,c,d],[c,d,e,f],Kombi).
Kombi = [[a, [c, d, e, f]], [b, [c, d, e, f]]]
```

1.6.2 Entwurf

Bei diesem Prädikat definieren wir uns zuerst einen einfachen Fall auf die sich unsere Rekursion bei komplizierteren Listen stützt.

Wenn die `InfoListe` leer ist, geben wir eine leere Liste aus unabhängig von den Daten.

Für die Rekursion, wenn die `InfoListe` mehr-elementig ist, unterscheiden wir zwischen zwei Fällen:

1. Wenn der Kopf der `InfoListe` in `Daten` vorhanden ist (`member`) rufen wir rekursiv das Prädikat `kombiniere` auf den Rest der `InfoListe` auf, und geben das Ergebnis aus.
2. Wenn der Kopf der `InfoListe` nicht in `Daten` vorhanden ist rufen wir rekursiv das Prädikat `kombiniere` auf den Rest der `InfoListe` auf, und kombinieren das Ergebnis per `[.|.]-Operator` mit der Verknüpfung des Kopfes der `InfoListe` und den `Daten`.

1.6.3 Quelltext

```
% kombiniere([a,b,c,d],[c,d,e,f],Kombi).
%   -> Kombi=[[a,[c,d,e,f]],[b,[c,d,e,f]]]
kombiniere([],_,[]):-!.
kombiniere([K|R],D,Out):- member(K,D),kombiniere(R,D,Out),!.
% in der nächsten Zeile gilt \+member wegen dem Cut
kombiniere([K|R],D,Out):- kombiniere(R,D,Tmp),Out=[[K,D]|Tmp],!.
```

1.7 zyklisch_succ

1.7.1 Aufgabenstellung

Schreiben Sie ein Prädikat `zyklisch_succ`, das Listen zyklisch um eine bestimmte Anzahl an Elementen verschiebt. **Beispiel:**

```
zyklisch_succ([a,b,c,d,e,f],3,Out).
Out = [d, e, f, a, b, c].
```

Als vordefiniertes Prädikat darf nur `append` verwendet werden!

1.7.2 Entwurf

Da die Aufgabenstellung etwas unklar ist, und das Beispiel nicht die volle Funktionalität des Prädikats darstellt gehen wir bei unserer Lösung davon aus, das folgende Beispiele zutreffen:

```
zyklisch_succ([a,b,c,d,e,f,g,h],3,Out).
Out = [d, e, f, a, b, c, g, h].
zyklisch_succ([a,b,c,d,e,f,g,h],2,Out).
Out = [c, d, a, b, g, h, e, f].
zyklisch_succ([a,b,c,d,e,f,g],2,Out).
Out = [c, d, a, b, g, e, f].
```

Den Lösungsweg den wir anstreben ist, dass Problem in mehrere kleinere Probleme zu zerlegen. Dazu definieren wir uns weitere Prädikate die wir in `zyklisch_succ` nacheinander aufrufen.

Zuerst zerlegen wir die gegebene Liste in Teillisten, welche die mitgeteilte Anzahl an Elementen als maximum haben. So würde die Liste `[a, b, c, d, e]` mit der Anzahl 2 in folgende

Liste zerlegt werden: `[[a, b], [c, d], [e]]`. Die so erzeugte Liste hat den Effekt das wir jedes Tupel für die weitere Verarbeitung als einzelnes Element betrachten können unabhängig von der mitgeteilten Anzahl an Elementen.

Nach dem aufteilen kommt die eigentliche Funktionalität des Verschiebens. Dafür vertauschen wir jedes Element der Liste mit geraden (0,2,4,...) Index mit seinem Nachfolger mit ungeraden Index (1,3,5,...) falls vorhanden. Dadurch wird aus der Liste `[[a, b], [c, d], [e]]` folgende: `[[[c, d], [a, b]], [[e]]]`. Die Elemente sind nun in gewünschter Reihenfolge wenn auch noch verschachtelt.

Nun muss aus der verschachtelten Liste eine einfache werden. Dafür gibt es normalerweise das Prädikat `flatten`, das wir jedoch nicht verwenden dürfen. Da es ohne das Prädikat `atom` (zum Feststellen, ob etwas ein Element oder eine Liste ist) schlicht nicht möglich ist `flatten` zu implementieren, streben wir einen Ansatz an, der zumindest unsere Listenstruktur abflacht. Dafür planen wir eine Mischung aus `append` Prädikat und **[.].-Operator**. So wird aus der verschachtelten Liste `[[[c, d], [a, b]], [[e]]]` die Liste `[c, d, a, b, e]`, welche wir nur noch ausgeben müssen.

1.7.3 Quelltext

```
% zyklisch_succ(Liste, Anzahl, Ausgabe): List x Natural0 -> List
%   zyklisch_succ([a,b,c,d,e,f],3,Out). -> Out=[d,e,f,a,b,c].
zyklisch_succ(L,0,L):-!.
zyklisch_succ([K|R],I,Out):- append(R,[K],Tmp),J is I-1,
    zyklisch_succ(Tmp,J,Out),!.
```

1.7.4 Anmerkung

Die Auffassung der Aufgabenstellung in unserem Entwurf ist falsch. Die von uns gegebenen Beispiele müssen folgendermaßen aufgelöst werden:

```
zyklisch_succ([a,b,c,d,e,f,g,h],3,Out).
Out = [d,e,f,g,h,a,b,c].
zyklisch_succ([a,b,c,d,e,f,g,h],2,Out).
Out = [c,d,e,f,g,h,a,b].
zyklisch_succ([a,b,c,d,e,f,g],2,Out).
Out = [c,d,e,f,g,a,b].
```

Außerdem 2 weitere Beispiele:

```
zyklisch_succ([a,b,c],3,Out).
Out = [a,b,c].
zyklisch_succ([a,b,c],4,Out).
Out = [b,c,a].
```

Gelöst ist die Aufgabe dadurch, dass wir von der Eingabeliste solange den Kopf abschneiden und hinten an den Rest mit `append` dran hängen, bis die mitgeteilte Anzahl, die wir bei jedem Aufruf um 1 verringern, 0 ist. Anschließend ist die Eingabeliste um die mitgeteilte Anzahl nach Links verschoben und wird über die Ausgabe ausgegeben.

1.8 Der Nibelungen Not

1.8.1 Aufgabenstellung

Der Nibelungen Not: Brunhild hasst Siegfried, Gunther und Krimhild. Krimhild liebt Siegfried und hasst Brunhild. Siegfried liebt Krimhild und mag Gunther. Gunther liebt Brunhild und mag Krimhild und Hagen. Hagen hasst Siegfried und alle, die Siegfried lieben. Brunhild mag alle, die Siegfried hassen. Alberich hasst alle, mit Ausnahme von sich selbst.

- a. Schreiben Sie diese Aussagen als PROLOG-Programm.
- b. Stellen Sie folgende Fragen (als Kommentar in die Programmdatei aufnehmen!):
 - (1) Wer hasst Siegfried?
 - (2) Wen mag Brunhild?
 - (3) Wer hasst wen?
 - (4) Wer liebt wen?
- c. Definieren Sie ein Prädikat `ideales_paar(X, Y)`, das zutrifft, falls X von Y und Y von X geliebt wird?
Wer von den Nibelungen wäre ein ideales Paar (als Kommentar in die Programmdatei aufnehmen!)?

1.8.2 Entwurf

- a. Die primitiven Aussagen der ersten 4 Sätze werden mit den drei Fakten `liebt(X, Y)`, `hasst(X, Y)` und `mag(X, Y)` für jeden Fall definiert. Die Personen werden hierbei als Konstanten angegeben.
Die letzten drei Sätze könnte man auch für alle einzelnen Fälle manuell definieren, jedoch geht es auch einfacher, indem die bereits vorhandenen (und zukünftigen) Fakten genutzt werden, um allgemeine Regeln zu formulieren.
- b. Die jeweiligen Fragen sind mit richtig formulierten Fakten und Regeln leicht zu schreiben. Bei den ersten zwei Fragen wird das jeweilige Prädikat mit einer Variable und dem Namen der Person als Konstante aufgerufen. Bei den letzten beiden Fragen werden die Prädikate mit jeweils zwei Variablen aufgerufen, um alle Kombinationen aufgelistet zu bekommen.
- c. Das Prädikat `ideales_paar(X, Y)` wird durch das zweimalige aufrufen des `liebt` Prädikats mit jeweils vertauschten X und Y Argumenten definiert.

1.8.3 Quelltext

```
% hasst(X,Y) := X hasst Y
% Bsp: hasst(Wer,prolog). -> Wer = _G348.
hasst(brunhild,siegfried).%wg S1
hasst(brunhild,gunther). %wg S1
hasst(brunhild,krimhild). %wg S1
hasst(krimhild,brunhild). %wg S2
hasst(hagen,siegfried). %wg S5
hasst(hagen,X):- liebt(X,siegfried). %wg S5
```

```

hasst(alberich,X):- X\=alberich. %wg S7

% liebt(X,Y) := X liebt Y
liebt(krimhild,siegfried). %wg S2
liebt(siegfried,krimhild). %wg S3
liebt(gunther,brunhild). %wg S4

% mag(X,Y) := X mag Y
mag(siegfried,gunther). %wg S3
mag(gunther,krimhild). %wg S4
mag(gunther,hagen). %wg S4
mag(brunhild,X):-hasst(X,siegfried). %wg S6

/* Fragen:
(1) Wer hasst Siegfried?
1 ?- hasst(X,siegfried).
   Siegfried wird gehasst von: brunhild, hagen, alberich.
(2) Wen mag Brunhild?
2 ?- mag(brunhild,X).
   Brunhild mag: brunhild, hagen, alberich.
(3) Wer hasst wen?
3 ?- hasst(X,Y).
   brunhild hasst siegfried ;
   brunhild hasst gunther ;
   brunhild hasst krimhild ;
   krimhild hasst brunhild ;
   hagen hasst siegfried ;
   hagen hasst krimhild ;
folgende Kombinationen werden nicht ausgegeben:
   alberich hasst brunhild ;
   alberich hasst siegfried ;
   alberich hasst gunther ;
   alberich hasst krimhild ;
   alberich hasst hagen.
(4) Wer liebt wen?
4 ?- liebt(X,Y).
   krimhild liebt siegfried ;
   siegfried liebt krimhild ;
   gunther liebt brunhild.
*/

% ideales_paar(X,Y) := X bildet mit Y ein ideales Paar
ideales_paar(X,Y):- liebt(X,Y), liebt(Y,X).

```

1.8.4 Anmerkung

Die Frage `3 ?- hasst(X,Y)` gibt nicht alle möglichen Kombinationen aus, weil Aufgrund der kurzen Implementation von Satz 7 nicht erfragt werden kann wen Alberich hasst. Damit die Frage `3 ?- hasst(X,Y)` auch die Personen anzeigt die Alberich hasst, müsste für jede

Person P ein eigener Fakt der Form `hasst(alberich,P)` erstellt werden. Eine andere Lösung des Problems wäre die Definition eines Prädikats `person` die bei jeder Person zu `true` evaluiert, um die Anzahl der Personen klar einzugrenzen.

Letzteres sähe dann so aus:

```
person(brunhild).
person(siefgried).
person(gunther).
person(krimhild).
person(hagen).
person(alberich).
hasst(alberich,X):- person(X), X\=alberich.
```

Informationen zur Signatur

	Unterzeichner	EMAILADDRESS=robin.ladiges@haw-hamburg.de, CN=Robin Christopher Ladiges
	Datum/Zeit	Sun Jun 27 00:05:35 CEST 2010
	Austeller-Zertifikat	CN=CAcert Class 3 Root, OU=http://www.CAcert.org, O=CAcert Inc.
	Serien-Nr.	44727
	Methode	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signatur)