Prof. Dr. J. Giesl
M. Brockschmidt, F. Emmes, C. Otto, T. Ströder

Tutoraufgabe 1 (Programmieren in Prolog):

In dieser Aufgabe sollen einige Abhängigkeiten im Übungsbetrieb Programmierung in Prolog modelliert und analysiert werden. Die gewählten Personennamen sind frei erfunden und eventuelle Übereinstimmungen mit tatsächlichen Personennamen sind purer Zufall.

Person	Rang
J. Giesl (jgi)	Professor
M. Brockschmidt (mbr)	Assistent
F. Emmes (fem)	Assistent
C. Otto (cot)	Assistent
T. Ströder (tst)	Assistent
T. Janson (tja)	Tutor
O. Kautz (oka)	Tutor
F. Ail (fai)	Student
N. Erd (ner)	Student
M. Ustermann (mus)	Student

Schreiben Sie keine Prädikate außer den geforderten und nutzen Sie bei Ihrer Implementierung jeweils Prädikate aus den vorangegangenen Aufgabenteilen.

a) Übertragen Sie die Informationen der Tabelle in eine Wissensbasis für Prolog. Geben Sie hierzu Fakten für die Prädikatssymbole `person` und `hatRang` an. Hierbei gilt `person(X)`, falls `X` eine Person ist und `hatRang(X, Y)`, falls `X` den Rang `Y` hat.

b) Stellen Sie eine Anfrage an das im ersten Aufgabenteil erstellte Programm, mit der man herausfinden kann, wer ein Assistent ist.

Hinweise:

- Durch die wiederholte Eingabe von " ;" nach der ersten Antwort werden alle Antworten ausgegeben.

c) Schreiben Sie ein Prädikat `bossVon`, womit Sie abfragen können, wer innerhalb der Übungsbetriebshierarchie einen Rang direkt über dem eines anderen bekleidet. Die Reihenfolge der Ränge ist Professor $>$ Assistent $>$ Tutor $>$ Student. So ist z.B. `bossVon(mbr,tja)` wahr, während `bossVon(tst,fai)` und `bossVon(tja,mbr)` beide falsch sind.

d) Stellen Sie eine Anfrage, mit der Sie herausfinden, zu wem es jemand anderen gibt, der in der Übungsbetriebshierarchie genau eine Stufe darunter steht. Es soll bei jeder Antwort nur derjenige auf dem jeweils höheren Rang, nicht aber derjenige auf dem niedrigeren Rang ausgegeben werden. Mehrfache Antworten mit dem gleichen Ergebnis sind allerdings erlaubt.

e) Schreiben Sie nun ein Prädikat `hatGleichenRang` mit einer Regel (ohne neue Fakten), mit dem Sie alle Paare von Personen abfragen können, die den gleichen Rang innerhalb des Übungsbetriebs bekleiden. So ist z.B. `hatGleichenRang(mbr, tst)` wahr, während `hatGleichenRang(tja, fai)` falsch ist. Stellen Sie sicher, dass `hatGleichenRang(X, Y)` nur dann gilt, wenn `X` und `Y` Personen sind.

f) Schreiben Sie schließlich ein Prädikat `vorgesetzt` mit zwei Regeln (wieder ohne neue Fakten), mit dem Sie alle Paare von Personen abfragen können, sodass die erste Person in der Übungsbetriebshierarchie der zweiten Person vorgesetzt ist. Eine Person `X` ist einer Person `Y` vorgesetzt, wenn der Rang von `X` "größer" als der Rang von `Y` ist (wobei wieder Professor $>$ Assistent $>$ Tutor $>$ Student gilt). So sind z.B. `vorgesetzt(mbr, tja)` und `vorgesetzt(tst, fai)` beide wahr, während `vorgesetzt(tja, mbr)` falsch ist. Stellen Sie auch hier sicher, dass `vorgesetzt(X, Y)` nur dann gilt, wenn `X` und `Y` Personen sind.

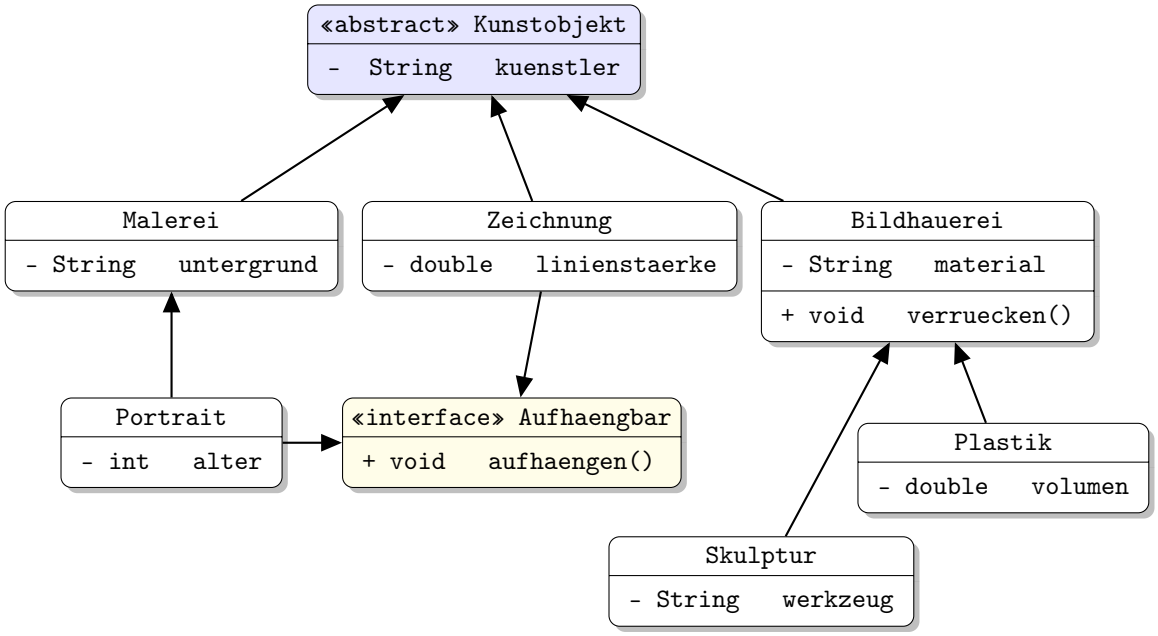
Lösung:

```
%Teilaufgabe a)
person(jgi).
person(mbr).
person(fem).
person(cot).
person(tst).
person(tja).
person(oka).
person(fai).
person(ner).
person(mus).

hatRang(jgi, professor).
hatRang(mbr, assistent).
hatRang(fem, assistent).
hatRang(cot, assistent).
hatRang(tst, assistent).
hatRang(tja, tutor).
hatRang(oka, tutor).
hatRang(fai, student).
hatRang(ner, student).
hatRang(mus, student).

%Teilaufgabe b)
%hatRang(X, assistent).
%Ausgabe:
%X = mbr ;
%X = fem ;
%X = cot ;
%X = tst.

%Teilaufgabe c)
bossVon(X, Y) :- hatRang(X, professor), hatRang(Y, assistent).
bossVon(X, Y) :- hatRang(X, assistent), hatRang(Y, tutor).
bossVon(X, Y) :- hatRang(X, tutor), hatRang(Y, student).

%Teilaufgabe d)
%bossVon(X, _).
%Ausgabe:
%X = jgi ;
%X = jgi ;
%X = jgi ;
%X = jgi ;
%X = mbr ;
%X = mbr ;
%X = fem ;
%X = fem ;
%X = cot ;
%X = cot ;
%X = tst ;
%X = tst ;
%X = tja ;
%X = tja ;
%X = tja ;
%X = oka ;
%X = oka ;
```

```
%X = ok ;
%false.

%Teilaufgabe e)
hatGleichenRang(X, Y) :- person(X), person(Y), hatRang(X, R), hatRang(Y, R).

%Teilaufgabe f)
vorgesetzt(B, S) :- bossVon(B, S), person(B), person(S).
vorgesetzt(B, S) :- bossVon(B, X), person(B), vorgesetzt(X, S).
```

Aufgabe 2 (Programmieren in Prolog): (2 + 1 + 2 + 2 = 7 Punkte)

Wir betrachten die Java-Klassenhierarchie aus der Präsenzübung:



a) Übertragen Sie die Klassenhierarchie in eine Wissensbasis für Prolog. Beschränken Sie sich hierbei auf die Namen der Klassen und des Interfaces. Verwenden Sie die zweistelligen Prädikate `extends` und `implements`, so dass `extends(A, B)` gilt wenn die Klasse `A` (direkt) die Klasse `B` erweitert (also `A extends B` im Quellcode stehen würde). Die Bedeutung von `implements` ist analog zu verstehen.

b) Schreiben Sie eine Anfrage, die alle Klassen als Antwort zurückgibt, die die Klasse `Kunstobjekt` direkt erweitern.

c) Schreiben Sie das zweistellige Prädikat `instanceof`. Der Aufruf `instanceof(A, B)` soll wahr sein, wenn

- `A` und `B` identisch sind, oder
- `A` eine Klasse bzw. Interface `M` direkt erweitert/implementiert und `instanceof(M, B)` gilt

d) Erweitern Sie die Wissensbasis um Fakten `nichtAbstract(X)` für alle Klassen `X`, die weder `abstract` noch ein `interface` sind.

Schreiben Sie anschließend das zweistellige Prädikat `instanzMoeglich`. Der Aufruf `instanzMoeglich(A, B)` soll wahr sein, wenn eine Java-Variable vom Typ `A` auf ein Objekt der Klasse `B` verweisen kann. Berücksichtigen Sie hier, dass es zwar Variablen vom Typ einer abstrakten Klasse oder eines Interfaces geben kann, aber in dieser Variable nur Objekte von nicht-abstrakten Klassen stehen können.

Lösung:

```
% a)
extends(malerei, kunstobjekt).
extends(zeichnung, kunstobjekt).
extends(bildhauerei, kunstobjekt).
extends(plastik, bildhauerei).
extends(skulptur, bildhauerei).
extends(portrait, malerei).
implements(portrait, aufhaengbar).
implements(zeichnung, aufhaengbar).

% b)
% ?- extends(X, kunstobjekt).
% X = malerei ;
% X = zeichnung ;
% X = bildhauerei ;

% c)
instanceof(A, A).
instanceof(A, Z) :- extends(A, M),    instanceof(M, Z).
instanceof(A, Z) :- implements(A, M), instanceof(M, Z).

% d)
nichtAbstract(malerei).
nichtAbstract(zeichnung).
nichtAbstract(portrait).
nichtAbstract(bildhauerei).
nichtAbstract(skultur).
nichtAbstract(plastik).

instanzMoeglich(A, Z) :- instanceof(Z, A), nichtAbstract(Z).
```

Tutoraufgabe 3 (Programmieren mit Listen in Prolog):

In dieser Aufgabe geht es darum, "**Bahnhof** zu verstehen". Ein Bahnhof hat eine begrenzte Anzahl $n \in \mathbb{N}$ von Gleisen und ein unbegrenzt langes Wartegleis. Die Idee ist, dass ein ankommender Zug sich hinten bei den wartenden Zügen einreiht. Wenn bislang kein Zug wartet, dann steht der neue Zug entsprechend vorne auf dem Wartegleis. Sobald im Bahnhof ein Gleis frei ist, fährt der Zug, der am längsten wartet, auf dieses Gleis. Auf jedem Gleis des Bahnhofs kann maximal ein Zug stehen. Auf dem Wartegleis können hingegen beliebig viele Züge stehen.
Implementieren Sie in dieser Aufgabe Datenstrukturen und Prädikate, um mit solchen Bahnhöfen zu arbeiten. Benutzen Sie für Züge ein einstelliges Funktionssymbol, so dass jeder Zug eine Zahl als Wert hat. Als Beispiel stehen hier die Terme `zug(1)` und `zug(2)` für die Züge Zug_1 und Zug_2.

a) Verwenden Sie für Bahnhöfe ein zweistelliges Funktionssymbol `bahnhof(Wartegleis, Gleise)`, so dass im ersten Argument das Wartegleis repräsentiert ist und das zweite Argument Informationen über die Gleise enthält. Verwenden Sie für die Terme in den beiden Argumenten jeweils die vordefinierten Listen. Geben Sie freie Gleise durch die Konstante `frei` an, belegte Gleise durch den Term des entsprechenden Zuges. Geben Sie für die von Ihnen gewählte Darstellung der Datenstruktur die Termdarstellung eines Bahnhofs B an, wobei

- der Zug Zug_3 auf dem Wartegleis steht
- der Bahnhof vier Gleise hat
- auf den Gleisen 2 und 4 die Züge Zug_1 und Zug_2 stehen und die anderen beiden Gleise frei sind

b) Schreiben Sie ein Prädikat `einfuegen(Zug, GleiseVorher, GleiseNachher)`, das den Zug `Zug` auf ein freies Gleis von `GleiseVorher` stellt, so dass die Gleise anschließend durch `GleiseNachher` repräsentiert werden.

Gehen Sie hierbei davon aus, dass `GleiseVorher` mindestens ein freies Gleis enthält. Ihr Prädikat sollte für Bahnhöfe mit beliebig vielen Gleisen verwendbar sein.

c) Schreiben Sie ein Prädikat `bewegen(bahnhof(W, G), bahnhof(WNeu, GNeu))`, das wartende Züge auf freie Gleise bewegt. Hierbei ist es egal, welches freie Gleis belegt wird. In jedem Schritt soll der Zug bewegt werden, der am längsten wartet. Die Auswertung des Prädikats ist beendet, wenn kein Gleis frei ist oder kein Zug mehr wartet. Sorgen Sie dafür, dass dann in `WNeu` und `GNeu` die entsprechend angepassten Werte von `W` bzw. `G` stehen.

Schreiben Sie hierfür erst ein Hilfsprädikat `keinPlatz(Gleise)`. Dieses Prädikat ist genau dann wahr, wenn die übergebene Gleisinformation `Gleise` nirgendwo die `frei`-Markierung enthält.

Weiterhin ist es nützlich, das bereits implementierte Prädikat `einfuegen` zu verwenden.

Angewendet auf den obigen Beispiel-Bahnhof B hat der Ergebnis-Bahnhof also ein leeres Wartegleis, die Züge Zug_1 und Zug_2 (nach wie vor) auf den Gleisen 2 und 4 und den ehemals wartenden Zug Zug_3 auf Gleis 1 oder auf Gleis 3 (wobei das andere Gleis weiterhin frei ist).

d) Schreiben Sie ein Prädikat `neuerZug(Zug, bahnhof(W, G), bahnhof(WNeu, GNeu))`. Hier soll der Zug `Zug` hinten auf dem Wartegleis eingereiht werden und anschließend so viele wartende Züge wie möglich auf freie Gleise bewegt werden. Implementieren Sie dieses Prädikat mit nur einer einzigen Regel (ohne Fakten) und nutzen Sie die bereits implementierten Prädikate.

Sie dürfen ein einfaches zusätzliches Hilfsprädikat implementieren und benutzen, um die Information des Wartegleises anzupassen.

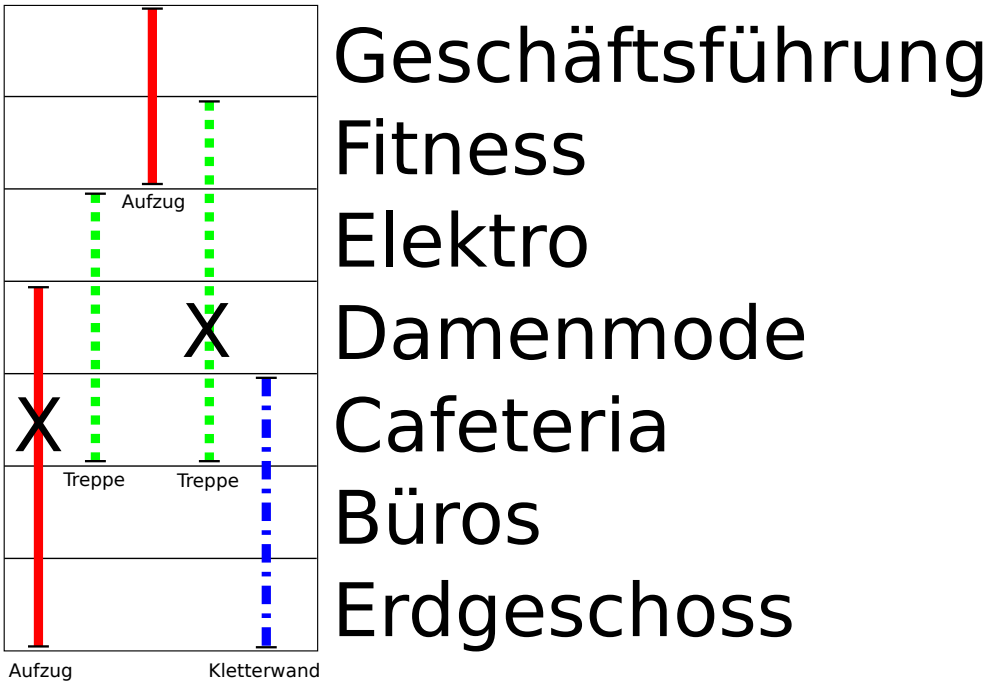
Lösung:

```
% a)
% Das Wartegleis wird durch eine Liste repraesentiert, wobei der am
% laengsten wartende Zug vorne steht.
% bahnhof([zug(3)], [frei, zug(1), frei, zug(2)]).

% b)
% Wenn das Gleis an der Stelle frei ist, stelle den Zug dort hin
einfuegen(Zug, [frei  |XS], [Zug   |XS]).
% Wenn das Gleis schon von einem Zug belegt ist, lasse diesen Zug da
% stehen und probiere es mit den restlichen Gleisen.
einfuegen(Zug, [zug(X)|XS], [zug(X)|ZS]) :- einfuegen(Zug, XS, ZS).

% c)
% Wenn kein Zug wartet, gibt es nichts zu tun
bewegen(bahnhof([], Gleise), bahnhof([], Gleise)).
% wenn kein Gleis frei ist, gibt es nichts zu tun
bewegen(bahnhof([X|XS], Gleise), bahnhof([X|XS], Gleise)) :-
    keinPlatz(Gleise).
bewegen(bahnhof([Zug|WartendRest], Gleise), bahnhof(WartendNeu, GleiseNeu)) :-
  % wir schaffen es, einen wartenden Zug auf ein Gleis zu stellen
    einfuegen(Zug, Gleise, GleiseTemp),
  % wir versuchen es weiter, bis nichts mehr geht
    bewegen(bahnhof(WartendRest, GleiseTemp), bahnhof(WartendNeu, GleiseNeu)).

keinPlatz([]).
keinPlatz([zug(_)|XS]) :- keinPlatz(XS).

% d)
neuerZug(Zug, bahnhof(Wartend, Gleise), bahnhof(WartendNeu, GleiseNeu)) :-
  % wir reihen den Zug in die Warteschlange ein
    hintenEinfuegen(Zug, Wartend, WartendTemp),
  % und bewegen, wenn es passt, wartende Zuege in den Bahnhof
    bewegen(bahnhof(WartendTemp, Gleise), bahnhof(WartendNeu, GleiseNeu)).

hintenEinfuegen(Zug, [], [Zug]).
hintenEinfuegen(Zug, [X|XS], [X|Ergebnis]) :-
    hintenEinfuegen(Zug, XS, Ergebnis).
```

Aufgabe 4 (Programmieren mit Listen in Prolog): (2 + 1 + 1 + 1 + 4 + 1 = 10 Punkte)

Wir beschäftigen uns in dieser Aufgabe mit den diversen Möglichkeiten, wie man in einem Hochhaus die einzelnen Etagen erreichen kann.
Eine der bekannteren Transportmöglichkeiten ist der **Aufzug**. Weiterhin gibt es **Treppen**, die verschiedene Etagen miteinander verbinden. Für einige Etagen wurde für Sportbegeisterte auch die Möglichkeit geschaffen, über **Kletterwände** andere Etagen zu erreichen.

a) Übertragen Sie die in der folgenden Grafik aufgeführten Verbindungen in eine Wissensbasis für Prolog. Geben Sie hierzu Fakten für das zweistellige Prädikatsymbol `verbindung` an. Hierbei gilt `verbindung(XS, A)`, falls die in der Liste `XS` aufgeführten Etagen mit der Verbindungsmöglichkeit `A` (also Aufzug, Treppe oder Kletterwand) verbunden sind. Hierbei soll die Liste die in der Grafik aufgeführte Reihenfolge berücksichtigen, so dass untere Etagen weiter vorne in der Liste sind. Beachten Sie auch, dass die mit X markierten Etagen von der jeweiligen Verbindung nicht erreicht werden.



b) Stellen Sie eine Anfrage an das im ersten Aufgabenteil erstellte Programm, mit der man herausfinden kann, welche Verbindungen sich über exakt drei (beliebige) erreichbare Etagen erstrecken.

Hinweise:

- Durch die wiederholte Eingabe von `";"` nach der ersten Antwort werden alle Antworten ausgegeben.

c) Schreiben Sie ein zweistelliges Prädikat `inListe`, so dass `inListe(XS, X)` wahr ist, wenn `X` in der Liste `XS` enthalten ist.

d) Schreiben Sie mit Hilfe von `inListe` ein dreistelliges Prädikat `weiterHinten`. Hierbei soll `weiterHinten(XS, A, B)` wahr sein, wenn `A` in der Liste `XS` vorhanden ist und `B` in der Liste `XS` weiter hinten auftaucht als `A`.

Es gilt also beispielsweise `weiterHinten([a, c, b, a], a, b)`, da das einzige `b` hinter dem ersten `a` steht. Analog gilt `weiterHinten([a, c, b, a], b, c)` nicht. Außerdem gilt `weiterHinten([b, a, b], a, b)` und `weiterHinten([b, a, b], b, a)`.

e) Schreiben Sie nun ein dreistelliges Prädikat `moeglich`, so dass ein Aufruf `moeglich(A, B, XS)` dann wahr ist, wenn man ausschließlich unter Verwendung der in der Liste `XS` aufgeführten Transportmöglichkeiten von der Etage `A` zu der Etage `B` gelangen kann. Hierbei darf man immer nur nach oben fahren/steigen/klettern, allerdings ist es erlaubt, zwischendurch umzusteigen.

Mit der im ersten Aufgabenteil erzeugten Wissensbasis soll also beispielsweise die Anfrage `moeglich(erdgeschoss, geschaeftsfuehrung, [treppe, aufzug])` wahr sein, da die folgende Verbindung existiert:

1. `erdgeschoss` zu `damenmode` mit Aufzug
2. `damenmode` zu `elektro` mit Treppe
3. `elektro` zu `fitness` mit Treppe
4. `fitness` zu `geschaeftsfuehrung` mit Aufzug

f) Stellen Sie eine Anfrage, mit der man herausfinden kann, welche Etagen man erreichen kann, wenn man in der Etage `erdgeschoss` startet und ausschließlich Aufzüge und Kletterwände benutzt.

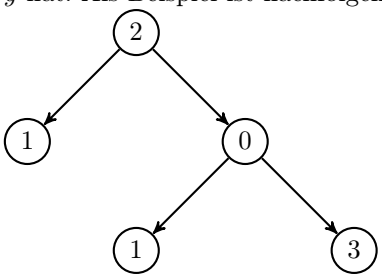
Lösung:

```
% a)
verbindung([erdgeschoss, bueros, damenmode], aufzug).
verbindung([cafeteria, damenmode, elektro],  treppe).
verbindung([fitness, geschaeftsfuehrung],    aufzug).
verbindung([cafeteria, elektro, fitness],    treppe).
verbindung([erdgeschoss, bueros, cafeteria], kletterwand).

% b)
% ?- verbindung([X, Y, Z], A).

% c)
inListe([X|_], X).
inListe([_|XS], Y) :- inListe(XS, Y).

% d)
weiterHinten([X|XS], A, B) :- weiterHinten(XS, A, B).
weiterHinten([A|XS], A, B) :- inListe(XS, B).

% e)
moeglich(X, X, _).
moeglich(START, ENDE, TYPEN) :- % beliebige Verbindung
                                verbindung(ETAGEN, TYP),
                                % mit passendem Typ
                                inListe(TYPEN, TYP),
                                % beliebige erreichbare hoehere Etage TEMP
                                weiterHinten(ETAGEN, START, TEMP),
                                % Rest der Strecke
                                moeglich(TEMP, ENDE, TYPEN).

% f)
% ?- moeglich(erdgeschoss, X, [aufzug, kletterwand])
% X = erdgeschoss ;
% X = bueros ;
% X = damenmode ;
% X = cafeteria ;
% X = damenmode ;
```

```
% X = bueros ;
% X = damenmode ;
% X = cafeteria ;
% X = cafeteria ;
```

Tutoraufgabe 5 (Prolog mit eigenen Datenstrukturen):

Natürliche Zahlen lassen sich in Prolog (oder anderen deklarativen Sprachen) durch die Peano-Notation als Terme darstellen. Dabei stellt die Konstante `0` die Zahl 0 dar und für eine Zahl x dargestellt durch den Term `X` stellt `s(X)` die Zahl $x+1$ dar. So wird z. B. die Zahl 3 durch den Term `s(s(s(0)))` dargestellt.
Binäre Bäume können in Prolog folgendermaßen als Terme dargestellt werden. Sei n eine natürliche Zahl dargestellt durch den Term `N`. Dann repräsentiert der Term `leaf(N)` einen Baum mit nur einem Blatt, welches den Wert n enthält. Für zwei Bäume x und y dargestellt durch die Terme `X` und `Y` repräsentiert der Term `node(X,N,Y)` einen binären Baum mit einem Wurzelknoten, der den Wert n enthält und die Teilbäume x und y hat. Als Beispiel ist nachfolgend ein binärer Baum und seine Darstellung als Term angegeben.

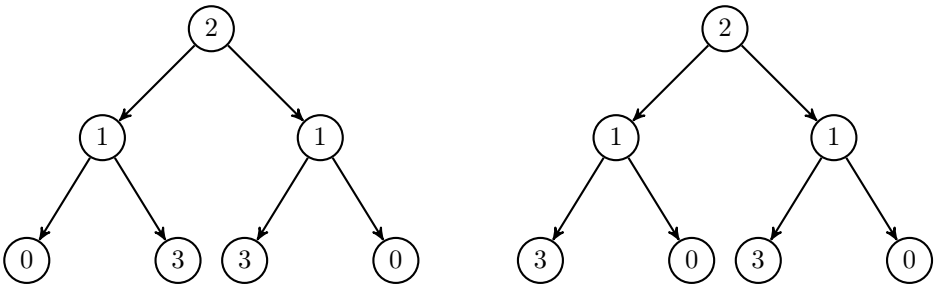



```
node(leaf(s(0)),s(s(0)),node(leaf(s(0)),0,leaf(s(s(s(0))))))
```

a) Schreiben Sie ein Prädikat `add/3` in Prolog, wobei `add(X,Y,Z)` genau dann wahr sein soll, wenn `X`, `Y` und `Z` natürliche Zahlen x, y und z in Peano-Notation repräsentieren und $x+y=z$ gilt.

b) Schreiben Sie ein Prädikat `leaves/2` in Prolog, wobei `leaves(X,Y)` genau dann wahr sein soll, wenn `X` einen binären Baum x und `Y` eine natürliche Zahl y in Peano-Notation repräsentieren und x genau y Blätter hat.

c) Schreiben Sie ein Prädikat `isSymmetric/1` in Prolog, wobei `isSymmetric(X)` genau dann wahr sein soll, wenn `X` einen symmetrischen binären Baum darstellt. Ein binärer Baum ist symmetrisch, wenn er nur aus einem Blatt besteht oder die beiden Teilbäume der Wurzel gespiegelt sind. Z. B. ist der folgende linke binäre Baum symmetrisch, während es der rechte nicht ist.



Lösung:

```
% isNat(X) gilt gdw. X eine natuerliche Zahl in Peano-Notation
% repraesentiert
isNat(0).
isNat(s(X)) :- isNat(X).
```

```
% add(X,Y,Z) gilt gdw. X, Y und Z natuerliche Zahlen in
% Peano-Notation repraesentieren und Z die Summe von X und Y ist
add(0,Y,Y) :- isNat(Y).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

% leaves(X,Y) gilt gdw. X einen binaeren Baum und Y eine natuerliche
% Zahl in Peano-Notation repraesentieren und Y die Anzahl der
% Blaetter von X ist
leaves(leaf(_),s(0)).
leaves(node(X,_,Y),Z) :- leaves(X,A),
                         leaves(Y,B),
                         add(A,B,Z).

% isSymmetric(X) gilt gdw. X einen symmetrischen binaeren Baum
% repraesentiert
isSymmetric(leaf(_)).
isSymmetric(node(X,_,Y)) :- mirror(X,Y).

% mirror(X,Y) gilt gdw. X und Y gespiegelte binaere Baeume
% repraesentieren
mirror(leaf(X),leaf(X)).
mirror(node(A,B,C),node(D,B,E)) :- mirror(A,E),
                                   mirror(C,D).
```

Aufgabe 6 (Prolog mit eigenen Datenstrukturen): (2 + 3 + 4 = 9 Punkte)

Betrachten Sie erneut die Darstellungen von natürlichen Zahlen und binären Bäumen aus der vorigen Tutoraufgabe.

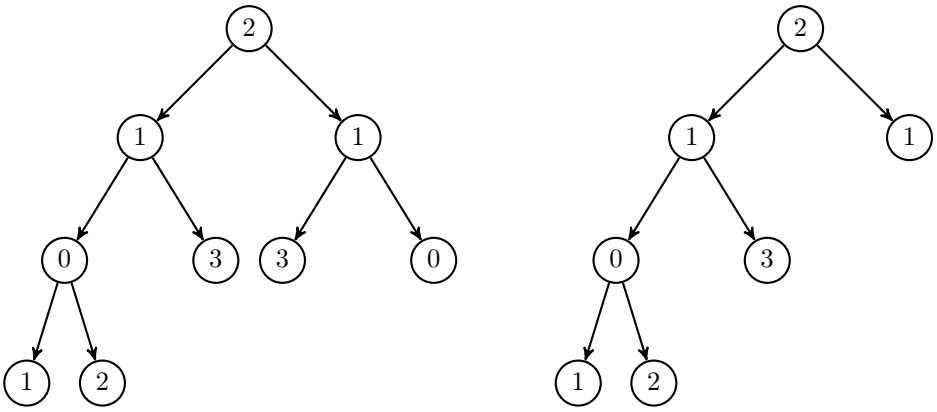
a) Schreiben Sie ein Prädikat `maxNeighbors/3`, wobei `maxNeighbors(X,Y,Z)` genau dann wahr sein soll, wenn `X`, `Y` und `Z` natürliche Zahlen x, y und z in Peano-Notation darstellen, z das Maximum von x und y ist und sich x und y höchstens um 1 unterscheiden (also Nachbarn oder gleich sind).

b) Schreiben Sie ein Prädikat `treeSum/2`, wobei `treeSum(X,Y)` genau dann wahr sein soll, wenn `X` einen binären Baum x und `Y` eine natürliche Zahl y in Peano-Notation darstellen und die Summe der in x enthaltenen natürlichen Zahlen y ist.

Hinweise:

- Das Prädikat `add/3` aus der vorigen Tutoraufgabe könnte hilfreich sein.

c) Schreiben Sie ein Prädikat `isBalanced/1` in Prolog, wobei `isBalanced(X)` genau dann wahr sein soll, wenn `X` einen binären Baum darstellt, welcher höhenbalanciert ist. Ein binärer Baum ist höhenbalanciert, wenn sich an jedem seiner Knoten die Höhe der beiden Teilbäume höchstens um 1 unterscheidet. Die Höhe eines binären Baums ist der längste Pfad von der Wurzel bis zu einem Blatt (ein binärer Baum, welcher nur aus einem Blatt besteht, hat also die Höhe 0). Zur Illustration sind nachfolgend zwei binäre Bäume der Höhe 3 abgebildet, wobei der linke höhenbalanciert ist, während der rechte dies nicht ist.



Hinweise:

- Es könnte hilfreich sein, ein Hilfsprädikat `isBalancedWithHeight/2` zu schreiben, welches gleichzeitig zum Test auf Höhenbalanciertheit die Höhe des entsprechenden binären Baums berechnet. Außerdem könnte das Prädikat `maxNeighbors/3` aus der ersten Teilaufgabe hilfreich sein.

Lösung:

```
% maxNeighbors(X,Y,Z) gilt gdw. X, Y und Z natuerliche Zahlen in
% Peano-Notation repraesentieren, Z das Maximum von X und Y ist
% und sich X und Y hoechstens um 1 unterscheiden
maxNeighbors(0,0,0).
maxNeighbors(0,s(0),s(0)).
maxNeighbors(s(0),0,s(0)).
maxNeighbors(s(X),s(Y),s(Z)) :- maxNeighbors(X,Y,Z).

% treeSum(X,Y) gilt gdw. X einen binaeren Baum und Y eine
% natuerliche Zahl in Peano-Notation repraesentieren und Y die
% Summe der Eintraege in X ist
treeSum(leaf(X),X) :- add(0,X,X).
treeSum(node(X,Y,Z),N) :- treeSum(X,A),
                          treeSum(Z,B),
                          add(A,B,C),
                          add(C,Y,N).

% isBalanced(X) gilt gdw. X einen hoehenbalancierten binaeren Baum
% repraesentiert (dies ist der Fall, wenn X einen
% hoehenbalancierten binaeren Baum mit einer beliebigen Hoehe
% repraesentiert)
isBalanced(X) :- isBalancedWithHeight(X,_).

% isBalancedWithHeight(X,Y) gilt gdw. X einen hoehenbalancierten
% binaeren Baum und Y eine natuerliche Zahl in Peano-Notation
% repraesentieren und Y die Hoehe von X ist
isBalancedWithHeight(leaf(_),0).
isBalancedWithHeight(node(X,_,Y),s(Z)) :- isBalancedWithHeight(X,A),
                                          isBalancedWithHeight(Y,B),
                                          % wenn beide Teilbaeume
                                          % hoehenbalanciert sind,
                                          % duerfen deren Hoehen
                                          % sich hoechstens um 1
                                          % unterscheiden und die
```

```
% Hoehe des gesamten Baums
% ist ihr Maxmimum plus 1
maxNeighbors(A,B,Z).
```