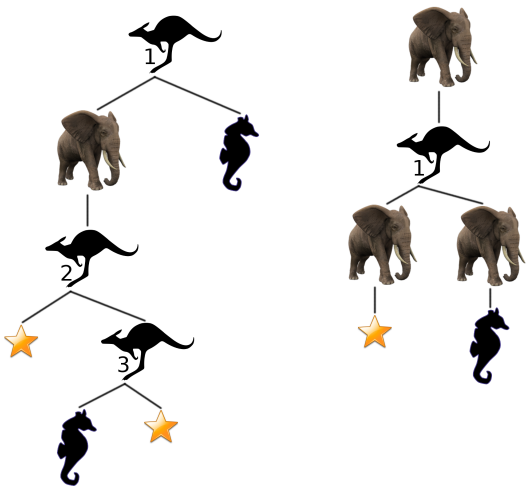
Prof. Dr. M. Müller C. Aschermann, J. Hensel, J. Protze, P. Reble

Allgemeine Hinweise:

- Die **Hausaufgaben** sollen in Gruppen von je **3 Studierenden** aus der **gleichen Kleingruppenübung (Tutorium)** bearbeitet werden. **Namen und Matrikelnummern** der Studierenden sind auf jedes Blatt der Abgabe zu schreiben. **Heften bzw. tackern Sie die Blätter!**
- Die **Nummer der Übungsgruppe** muss **links oben** auf das **erste Blatt** der Abgabe geschrieben werden. Notieren Sie die Gruppennummer gut sichtbar, damit wir besser sortieren können.
- Die Lösungen müssen **bis Montag, den 01.02.2016 um 15:00 Uhr** in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55). Alternativ können Sie die Lösungen auch vor der Abgabefrist direkt bei Ihrer Tutorin/Ihrem Tutor abgeben.
- In einigen Aufgaben müssen Sie in Haskell oder Prolog programmieren und `.hs`- bzw. `.pl`-Dateien anlegen. **Drucken** Sie diese aus **und** schicken Sie sie per **E-Mail** vor Montag, dem 01.02.2016 um 15:00 Uhr an Ihre Tutorin/Ihren Tutor.
  Stellen Sie sicher, dass Ihr Programm von `GHC` bzw. `SWI` akzeptiert wird, ansonsten werden keine Punkte vergeben.

Tutoraufgabe 1 (Datenstrukturen in Haskell):

In dieser Aufgabe beschäftigen wir uns mit *Kindermobiles*, die man beispielsweise über das Kinderbett hängen kann. Ein Kindermobile besteht aus mehreren Figuren, die mit Fäden aneinander aufgehängt sind. Als mögliche Figuren im Mobile beschränken wir uns hier auf *Sterne*, *Seepferdchen*, *Elefanten* und *Kängurus*.
An Sternen und Seepferdchen hängt keine weitere Figur. An jedem Elefant hängt eine weitere Figur, unter jedem Känguru hängen zwei Figuren. Weiterhin hat jedes Känguru einen Beutel, in dem sich etwas befinden kann (z. B. eine Zahl).
In der folgenden Grafik finden Sie zwei beispielhafte Mobiles[1].



[1] Für die Grafik wurden folgende Bilder von Wikimedia Commons verwendet:

- Stern `http://commons.wikimedia.org/wiki/File:Crystal_Clear_action_bookmark.png`
- Seepferdchen `http://commons.wikimedia.org/wiki/File:Seahorse.svg`
- Elefant `http://commons.wikimedia.org/wiki/File:African_Elephant_Transparent.png`
- Känguru `http://commons.wikimedia.org/wiki/File:Kangourou.svg`

a) Entwerfen Sie einen parametrischen Datentyp `Mobile a` mit vier Konstruktoren (für Sterne, Seepferdchen, Elefanten und Kängurus), mit dem sich die beschriebenen Mobiles darstellen lassen. Verwenden Sie den Typparameter `a` dazu, den Typen der Känguru-Beutelinhalte festzulegen.

Modellieren Sie dann die beiden oben dargestellten Mobiles als Ausdruck dieses Datentyps in Haskell. Nehmen Sie hierfür an, dass die gezeigten Beutelinhalte vom Typ `Int` sind.

Hinweise:

- Für Tests der weiteren Teilaufgaben bietet es sich an, die beiden Mobiles als konstante Funktionen im Programm zu deklarieren.
- Schreiben Sie `deriving Show` an das Ende Ihrer Datentyp-Deklaration. Damit können Sie sich in `GHCi` ausgeben lassen, wie ein konkretes Mobile aussieht.

```
mobileLinks :: Mobile Int
mobileLinks = ...
```

b) Schreiben Sie eine Funktion `count :: Mobile a -> Int`, die die Anzahl der Figuren im Mobile berechnet. Für die beiden gezeigten Mobiles soll also 8 und 6 zurückgegeben werden.

c) Schreiben Sie eine Funktion `liste :: Mobile a -> [a]`, die alle in den Känguru-Beuteln enthaltenen Elemente in einer Liste (mit beliebiger Reihenfolge) zurückgibt. Für das linke Mobile soll also die Liste `[1,2,3]` (oder eine Permutation davon) berechnet werden.

d) Schreiben Sie eine Funktion `greife :: Mobile a -> Int -> Mobile a`. Diese Funktion soll für den Aufruf `greife mobile n` die Figur mit Index `n` im Mobile `mobile` zurückgeben.

Wenn man sich das Mobile als Baumstruktur vorstellt, werden die Indizes entsprechend einer *Tiefensuche*[2] berechnet:

Wir definieren, dass die oberste Figur den Index 1 hat. Wenn ein Elefant den Index n hat, so hat die Nachfolgefigur den Index $n + 1$.

Wenn ein Känguru den Index n hat, so hat die linke Nachfolgefigur den Index $n+1$. Wenn entsprechend dieser Regeln alle Figuren im linken Teil-Mobile einen Index haben, hat die rechte Nachfolgefigur den nächsthöheren Index.

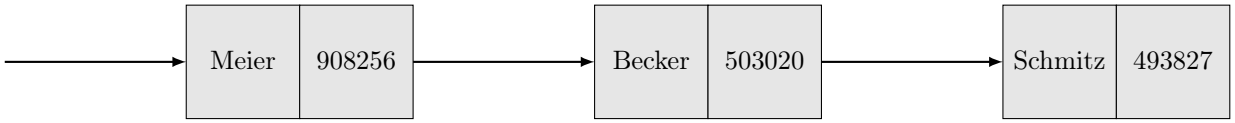
Im linken Beispiel-Mobile hat das Känguru mit Beutelinhalt `3` also den Index 5.

Hinweise:

- Benutzen Sie die Funktion `count` aus Aufgabenteil b).
- Falls der übergebene Index kleiner 1 oder größer der Anzahl der Figuren im Mobile ist, darf sich Ihre Funktion beliebig verhalten.

Aufgabe 2 (Datenstrukturen in Haskell): (1 + 1 + 1 + 2 + 2 = 7 Punkte)

In dieser Aufgabe betrachten wir eine einfache assoziative Datenstruktur in Form einer Liste.
Bei einer assoziativen Datenstruktur ist es möglich, einzelnen Schlüsselwerten jeweils einen Wert zuzuordnen. Beispielsweise kann jedes Telefonbuch als assoziative Datenstruktur verstanden werden, bei der zu jedem Namen (dem Schlüssel) im Buch die dazugehörige Telefonnummer (der Wert) angegeben ist.
Wir speichern in dieser Aufgabe die Paare aus Schlüssel und Wert in einer Liste. In der Beispielgrafik erkennt man eine solche Liste, die zum Namen `"Meier"` die Nummer `908256`, zum Namen `"Becker"` die Nummer `503020` und zum Namen `"Schmitz"` die Nummer `493827` gespeichert hat.



Verwenden Sie in dieser Aufgabe keine vordefinierten Listen oder Tupel!

[2] siehe auch Wikipedia: `http://de.wikipedia.org/wiki/Tiefensuche`

a) Entwerfen Sie einen parametrisierten Datentyp `AList value`, mit dem assoziative Listen über Schlüssel vom Datentyp `String` und Werte vom Datentyp `value` dargestellt werden können.

Hinweise:

- Für die nachfolgenden Aufgaben ist es sehr hilfreich, die obige Beispielliste unter dem Namen `bsp` zur Verfügung zu haben:

```
bsp :: AList Int
bsp = ...
```

- Ergänzen Sie `deriving Show` am Ende der Datenstruktur, damit `GHCi` die Listen auf der Konsole anzeigen kann: `data ... deriving Show`

b) Schreiben Sie die Funktion `size`, die eine assoziative Liste vom Typ `AList a` übergeben bekommt und als Rückgabe die Anzahl der Einträge in dieser Liste (also die Länge der Liste) als `Int` zurückgibt.

Für die Beispielliste (angenommen diese ist als `bsp` verfügbar) soll für den Aufruf `size bsp` also `3` zurückgegeben werden.

c) Schreiben Sie die Funktion `contained`. Diese bekommt als erstes Argument eine assoziative Liste vom Typ `AList a` und als zweites Argument einen `String`. Der Rückgabewert dieser Methode ist vom Typ `Bool`. Die Methode gibt `True` zurück, wenn in der Liste ein Element enthalten ist, das als Schlüssel den `String` des zweiten Arguments hat. Ansonsten wird `False` zurückgegeben.

Für die Beispielliste `bsp` soll der Aufruf `contained bsp "Becker"` also `True` zurückgeben, `contained bsp "Meyer"` gibt `False`.

d) Schreiben Sie die Funktion `put`. Diese bekommt als erstes Argument eine assoziative Liste vom Typ `AList a`, als zweites Argument einen `String` und als drittes Argument einen Wert vom Typ `a`. Der Rückgabewert der Funktion ist die wie folgt modifizierte assoziative Liste vom Typ `AList a`: Falls für den im zweiten Argument übergebenen Schlüssel bereits ein Element in der Liste existiert, wird dessen Wert mit dem im dritten Argument übergebenen Wert überschrieben. Falls kein solches Element existiert, wird dieses hinzugefügt.

Gehen Sie hierbei davon aus, dass zu jedem Schlüssel in der Liste auch nur genau ein Listenelement existiert.

Für die Beispielliste `bsp` soll der Aufruf `put bsp "Becker" 1` also die assoziative Liste zurückgeben, bei der statt `503020` nun `1` für den Schlüssel `"Becker"` gespeichert ist.

e) Schreiben Sie, analog zu `map`, die Funktion `mapAList`. Diese bekommt als erstes Argument eine Funktion vom Typ `a -> b` und als zweites Argument eine assoziative Liste vom Typ `AList a`. Der Rückgabewert der Funktion ist die wie folgt definierte assoziative Liste vom Typ `AList b`: Für jedes Paar `(x, y)` in der Eingabeliste (des zweiten Arguments) enthält die Ergebnisliste ein Paar `(x, z)`. Wenn `f` die Funktion des ersten Arguments ist, ist `z` das Ergebnis von `f y`. Die Ergebnisliste enthält keine weiteren Paare.

Für die Beispielliste `bsp` soll der Aufruf `mapAList (* (-1)) bsp` also die assoziative Liste zurückgeben, bei der Elemente für die Paare `(Meier, -908256)`, `(Becker, -503020)` und `(Schmitz, -493827)` vorhanden sind.

Tutoraufgabe 3 (Typen in Haskell):

Bestimmen Sie zu den folgenden Haskell-Funktionen `f`, `g`, `h` und `i` den jeweils allgemeinsten Typ. Geben Sie den Typ an und begründen Sie Ihre Antwort. Gehen Sie hierbei davon aus, dass alle Zahlen den Typ `Int` haben und die Funktion `+` den Typ `Int -> Int -> Int` hat.

i)
```
f []     x y = y
f [z:zs] x y = f [] (z:x) y
```

ii)
```
g x 1 = 1
g x y = (\x -> (g x 0)) y
```

iii)
```
data T a b = C0 | C1 a | C2 b | C3 (T a b) (T a b)

i (C3 (C1 [])     (C2 y)) = C1 0
i (C3 (C2 [])     (C1 y)) = C2 0
i (C3 (C1 (x:xs)) (C2 y)) = i (C3 (C1 y) (C2 [x]))
```

Hinweise:

- Versuchen Sie, diese Aufgabe ohne Einsatz eines Rechners zu lösen. Bedenken Sie, dass Sie in einer Prüfung ebenfalls keinen Rechner zur Verfügung haben.

Aufgabe 4 (Typen in Haskell): (1 + 1 + 2 + 3 = 7 Punkte)

Bestimmen Sie zu den folgenden Haskell-Funktionen `f`, `g`, `h` und `i` den jeweils allgemeinsten Typ. Geben Sie den Typ an und begründen Sie Ihre Antwort. Gehen Sie hierbei davon aus, dass alle Zahlen den Typ `Int` haben und die Funktionen `+`, `head` und `==` die Typen `Int -> Int -> Int`, `[a] -> a` und `a -> a -> Bool` haben.

i)
```
f (x : xs) y z = x + y
f xs       y z = if xs == z then y else head xs + y
```

ii)
```
g [] = g [1]
g x  = (\x -> x) x
```

iii)
```
h w x [] z = if z == [x] then w else h w x [] z
h w x y  z = if x then head y else (x, x)
```

iv)
```
data N a b = A a | F (a -> b) | I Int

i (F f) x = f x
i (A y) x = i k x
  where
    k = F (\x -> I y)
```

Hinweise:

- Versuchen Sie, diese Aufgabe ohne Einsatz eines Rechners zu lösen. Bedenken Sie, dass Sie in einer Prüfung ebenfalls keinen Rechner zur Verfügung haben.

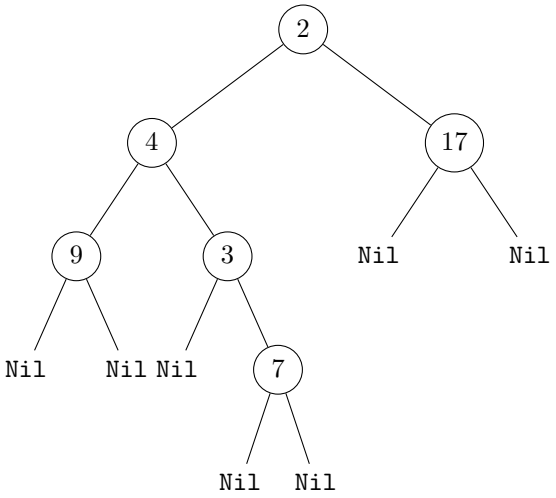
Tutoraufgabe 5 (Higher Order):

Wir betrachten Operationen auf dem Typ `Tree`, der (mit einem Testwert) wie folgt definiert ist:

```
data Tree = Nil | Node Int Tree Tree deriving Show

testTree = Node 2
 (Node 4 (Node 9 Nil Nil) (Node 3 Nil (Node 7 Nil Nil)))
 (Node 17 Nil Nil)
```

Man kann den Baum auch graphisch darstellen:



Wir wollen nun eine Reihe von Funktionen betrachten, die auf diesen Bäumen arbeiten:

```
decTree :: Tree -> Tree
decTree Nil = Nil
decTree (Node v l r) = Node (v - 1) (decTree l) (decTree r)

sumTree :: Tree -> Int
sumTree Nil = 0
sumTree (Node v l r) = v + (sumTree l) + (sumTree r)

flattenTree :: Tree -> [Int]
flattenTree Nil = []
flattenTree (Node v l r) = v : (flattenTree l) ++ (flattenTree r)
```

Wir sehen, dass diese Funktionen alle in der gleichen Weise konstruiert werden: Was die Funktion mit einem Baum macht, wird anhand des verwendeten Datenkonstruktors entschieden. Der nullstellige Konstruktor `Nil` wird einfach in einen Standard-Wert übersetzt. Für den dreistelligen Konstruktor `Node` wird die jeweilige Funktion rekursiv auf den Kindern aufgerufen und die Ergebnisse werden dann weiterverwendet, z.B. um ein neues `Tree`-Objekt zu konstruieren oder ein akkumuliertes Gesamtergebnis zu berechnen. Intuitiv kann man sich vorstellen, dass jeder Konstruktor durch eine Funktion der gleichen Stelligkeit ersetzt wird. Klarer wird dies, wenn man die folgenden alternativen Definitionen der Funktionen von oben betrachtet:

```
decTree' Nil = Nil
decTree' (Node v l r) = decN v (decTree' l) (decTree' r)
decN = \v l r -> Node (v - 1) l r

sumTree' Nil = 0
sumTree' (Node v l r) = sumN v (sumTree' l) (sumTree' r)
sumN = \v l r -> v + l + r

flattenTree' Nil = []
flattenTree' (Node v l r) = flattenN v (flattenTree' l) (flattenTree' r)
flattenN = \v l r -> v : l ++ r
```

Die definierenden Gleichungen für den Fall, dass der betrachtete Baum durch den Konstruktor `Node` erzeugt wird, kann man in allen diesen Definitionen so lesen, dass die eigentliche Funktion rekursiv auf die Kinder angewandt wird und der Konstruktor `Node` durch die jeweils passende Hilfsfunktion (`decN`, `sumN`, `flattenN`) ersetzt wird. Der Konstruktor `Nil` wird analog durch eine nullstellige Funktion (also eine Konstante) ersetzt. Als Beispiel kann der Ausdruck `decTree' testTree` dienen, der dem folgenden Ausdruck entspricht:

```
decN 2
 (decN 4 (decN 9 Nil Nil) (decN 3 Nil (decN 7 Nil Nil)))
 (decN 17 Nil Nil)
```

Im Baum `testTree` sind also alle Vorkommen von `Node` durch `decN` und alle Vorkommen von `Nil` durch `Nil` ersetzt worden.
Analog ist `sumTree' testTree` äquivalent zu

```
sumN 2
 (sumN 4 (sumN 9 0 0) (sumN 3 0 (sumN 7 0 0)))
 (sumN 17 0 0)
```

Im Baum `testTree` sind also alle Vorkommen von `Node` durch `sumN` und alle Vorkommen von `Nil` durch `0` ersetzt worden.
Die *Form* der Funktionsanwendung bleibt also gleich, nur die Ersetzung der Konstruktoren durch Hilfsfunktionen muss gewählt werden. Dieses allgemeine Muster wird durch die Funktion `foldTree` beschrieben:

```
foldTree :: (Int -> a -> a -> a) -> a -> Tree -> a
foldTree f c Nil = c
foldTree f c (Node v l r) = f v (foldTree f c l) (foldTree f c r)
```

Bei der Anwendung ersetzt `foldTree` alle Vorkommen des Konstruktors `Node` also durch die Funktion `f` und alle Vorkommen des Konstruktors `Nil` durch die Konstante `c`. Unsere Funktionen von oben können dann vereinfacht dargestellt werden:

```
decTree'' t = foldTree decN Nil t
sumTree'' t = foldTree sumN 0 t
flattenTree'' t = foldTree flattenN [] t
```

a) Implementieren Sie eine Funktion `prodTree`, die das Produkt der Einträge eines Baumes bildet. Es soll also `prodTree testTree = 2 * 4 * 9 * 3 * 7 * 17 = 25704` gelten. Verwenden Sie dazu die Funktion `foldTree`.

b) Implementieren Sie eine Funktion `incTree`, die einen Baum zurückgibt, in dem der Wert jeden Knotens um 1 inkrementiert wurde. Verwenden Sie dazu die Funktion `foldTree`.

Aufgabe 6 (Higher Order): (1 + 0.5 + 2.5 + 2 = 6 Punkte)

Wir betrachten Operationen auf dem parametrisierten Typ `List`, der (mit zwei Testwerten) wie folgt definiert ist:

```
data List a = Nil | Cons a (List a) deriving Show

testList, testList2 :: List Int
testList = Cons (-23) (Cons 42 (Cons 19 (Cons (-38) Nil)))
testList2 = Cons 1 (Cons 2 Nil)
```

Die Liste `testList` entspricht also $-23, 42, 19, -38$ und `testList2` der Liste $1, 2$.

a) Schreiben Sie eine Funktion `filterList :: (a -> Bool) -> List a -> List a`, die sich auf unseren selbstdefinierten Listen wie `filter` auf den vordefinierten Listen verhält. Es soll also die als erster Parameter übergebene Funktion auf jedes Element angewandt werden um zu entscheiden, ob dieses auch im Ergebnis auftaucht. Der Ausdruck `filterList (\x -> x > 30 || x < -30) testList` soll dann also zu `(Cons 42 (Cons -38 Nil))` auswerten.

b) Schreiben Sie eine Funktion `posList :: List Int -> List Int`, die die Teilliste der positiven Werte zurückgibt. Für `testList` soll also `Cons 42 (Cons 19 Nil)` zurückgegeben werden. Verwenden Sie dafür `filterList`.

c) Schreiben Sie eine Funktion `foldList :: (a -> b -> b) -> b -> List a -> b`, die wie `foldTree` aus der vorhergegangen Tutoraufgabe die Datenkonstruktoren durch die übergebenen Argumente ersetzt. Der Ausdruck `foldList f c (Cons` x_1 `(Cons` x_2 `...(Cons` x_n `Nil)...)` soll dann also äquivalent zu `(f` x_1 `(f` x_2 `...(f` x_n `c)...)` sein.

Beispielsweise soll für `times x y = x * y` der Ausdruck `foldList times 1 testList` zu `-23 * 42 * 19 * -38 = 697452` ausgewertet werden.

d) Implementieren Sie die Funktion `filterList` unter dem Namen `filterList'` erneut, verwenden Sie diesmal aber `foldList`, statt direkt Rekursion in der Definition von `filterList'` zu verwenden.

Tutoraufgabe 7 (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
M. Müller (mmu)	Professor
C. Aschermann (cas)	Assistent
J. Hensel (jhe)	Assistent
J. Protze (jpr)	Assistent
P. Reble (pre)	Assistent
S. Bender (sbe)	Tutor
I. Klöter (ikl)	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.

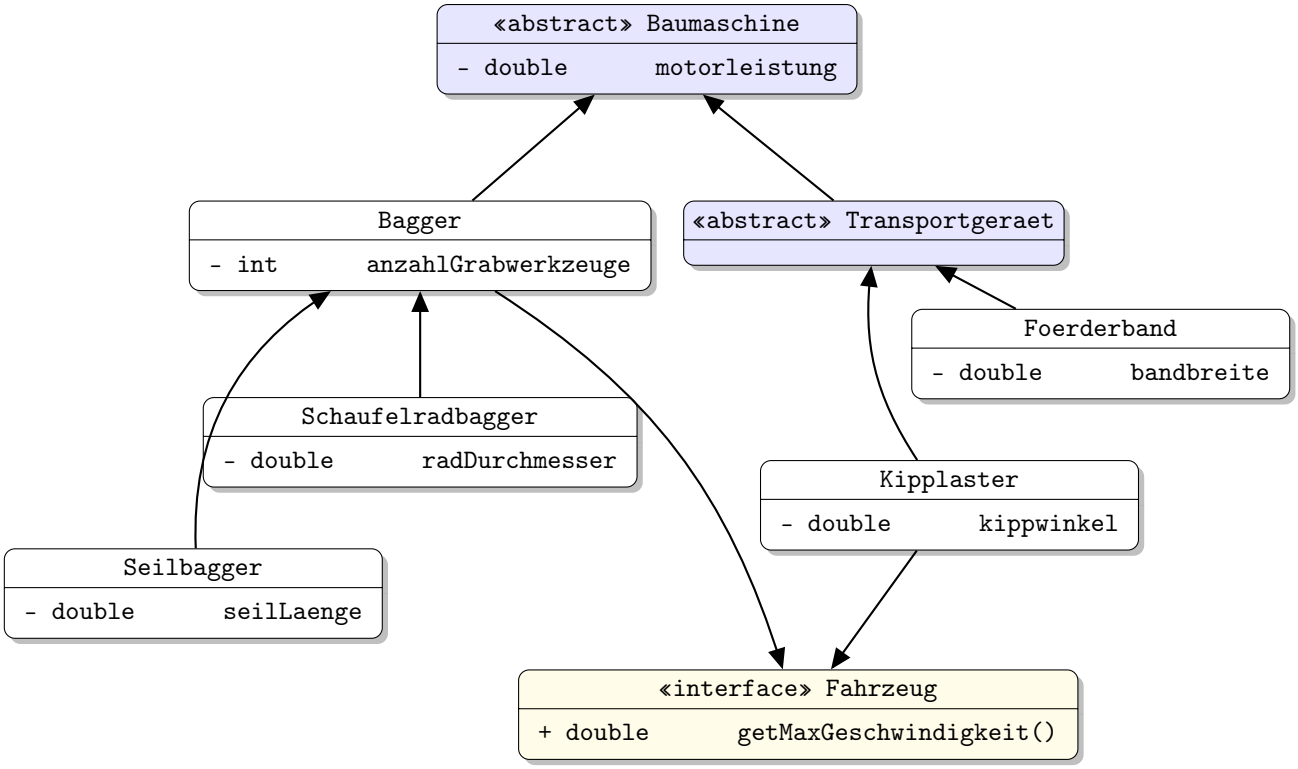
c) Schreiben Sie ein Prädikat `bossVon`, womit Sie abfragen können, wer innerhalb der Übungsbetriebshierarchie einen Rang direkt über dem eines anderen bekleidet. Die Reihenfolge der Ränge ist Professor $>$ Assistent $>$ Tutor $>$ Student. So ist z.B. `bossVon(cas,sbe)` wahr, während `bossVon(pre,fai)`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

d) 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(jpr, jhe)` wahr, während `hatGleichenRang(ikl, mmu)` falsch ist. Stellen Sie sicher, dass `hatGleichenRang(X, Y)` nur dann gilt, wenn `X` und `Y` Personen sind.

e) 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(pre, sbe)` und `vorgesetzt(cas, fai)` beide wahr, während `vorgesetzt(sbe, pre)` falsch ist. Stellen Sie auch hier sicher, dass `vorgesetzt(X, Y)` nur dann gilt, wenn `X` und `Y` Personen sind.

Aufgabe 8 (Programmieren in Prolog): (2 + 1 + 2 = 5 Punkte)

Wir betrachten die Java-Klassenhierarchie aus Übung 9:



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 `Transportgeraet` 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