Prof. Dr. J. Giesl

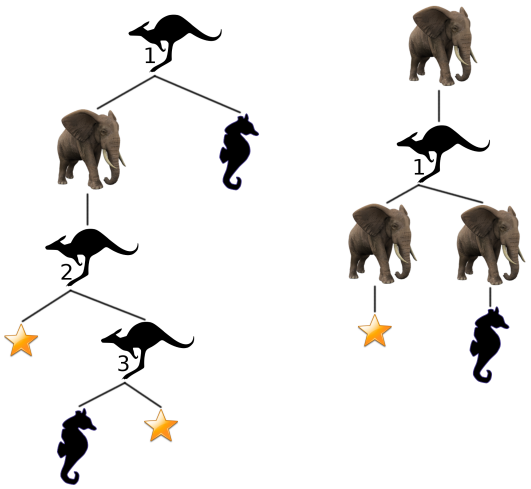
F. Frohn, J. Hensel, D. Korzeniewski

Allgemeine Hinweise:

- Die **Hausaufgaben** sollen in Gruppen von je **2 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 06.02.2017 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 oder unmittelbar vor Beginn in der Globalübung 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 06.02.2017 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.

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

```
mobileRight:: Mobile Int
mobileRight = ...
```

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.

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): (2 + 2 + 1.5 + 5 + 1.5 = 12 Punkte)

In dieser Aufgabe betrachten wir eine spezielle Art von Mehrwegbäumen. In diesen Bäumen gibt es zwei Arten von Knoten. Die erste Art sind *Index*-Knoten. Diese speichern zwei Werte und können eine beliebige Anzahl Nachfolger haben (auch 0 Nachfolger sind möglich). Die zweite Art sind *Daten*-Knoten. Diese speichern nur einen Wert und sind immer Blätter.
Der Baum `t1` in Abbildung 1 ist ein solcher Baum.

a) Entwerfen Sie einen parametrischen Datentyp `MultTree a`, der zur Darstellung der oben beschriebenen Bäume genutzt werden kann.

Hinweise:

- Für die nachfolgenden Aufgaben ist es hilfreich, sich den Baum `t1` als Konstante zu definieren:
  ```
  t1 :: MultTree Int
  t1 = ...
  ```

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

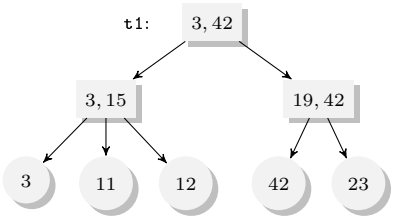


Abbildung 1: Ein Mehrwegbaum

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

b) Schreiben Sie eine Funktion `verzweigungsgrad`. Diese bekommt einen Mehrwegbaum vom Typ `MultTree a` übergeben und berechnet, wie viele Nachfolger ein Knoten in diesem Baum maximal hat.

Für den Beispielbaum würde der Aufruf `verzweigungsgrad t1` also den Rückgabewert 3 liefern.

c) Schreiben sie eine Funktion `datenListe`, die alle in den Daten-Knoten des übergebenen Baumes gespeicherten Werte in einer Liste zurück gibt.

Der Aufruf `datenListe t1` würde also `[3, 11, 12, 42, 23]` ergeben.

d) Schreiben Sie eine Funktion `datenIntervalle`, die einen Baum des Typs `MultTree Int` übergeben bekommt. Diese verändert die Index-Knoten so, dass der kleinere der beiden darin gespeicherten Werte dem kleinsten Wert entspricht, der in einem Daten-Knoten in diesem Teilbaum gespeichert ist. Analog soll der größere der beiden Werte dem größten Wert in einem Daten-Knoten in diesem Teilbaum entsprechen.

Für einen (Teil-)Baum ohne Daten-Knoten soll das Intervall `maxBound, minBound` in die Index-Knoten geschrieben werden.

Der Aufruf `datenIntervalle t1` würde also einen Baum zurückgeben, der sich von `t1` nur dadurch unterscheidet, dass `15` und `19` durch `12` bzw. `23` ersetzt sind.

Hinweise:

Die vordefinierten Konstanten `minBound :: Int` und `maxBound :: Int` liefern den kleinsten und den größten möglichen Wert vom Typ `Int`.

e) Schreiben Sie eine Funktion `contains`, die überprüft, ob ein gegebener Wert in einem Baum vom Typ `MultTree Int` enthalten ist. Gehen Sie dabei davon aus, dass alle Bäume, die der Funktion übergeben werden, bereits das Format haben, das die Funktion `datenIntervalle` erzeugt. Ein Knoten mit den Werten x, y hat also nur solche Blätter als Nachfolger, deren Wert im Intervall $[x, y]$ liegt. Nutzen sie dies, um ihre Implementierung effizienter zu gestalten!

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)
```
h (x:xs) y z = if x then h xs x (y:z) else h xs y z
```

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

i (C3 (C1 x)      (C2 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): (2 + 2 + 2 = 6 Punkte)

Bestimmen Sie zu den folgenden Haskell-Funktionen `f`, `g` und `h` 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 `+`/`-`, `length`, `==` und `>` die Typen `Int -> Int -> Int`, `[a] -> Int`, `a -> a -> Bool` und `Int -> Int -> Bool` haben.

i)
```
f 1 ys _ = ys
f x (y:ys) z = if x > z then f (x - 1) (x:ys) z else (y:ys)
```

ii)
```
g x (y:ys) = g (y x) ys
g x y      = x []
```

iii)
```
h [] x y = if x == 1 then h y (length y) [] else True
h (a:as) x y = h as (x + a) 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 (Funktionen höherer Ordnung):

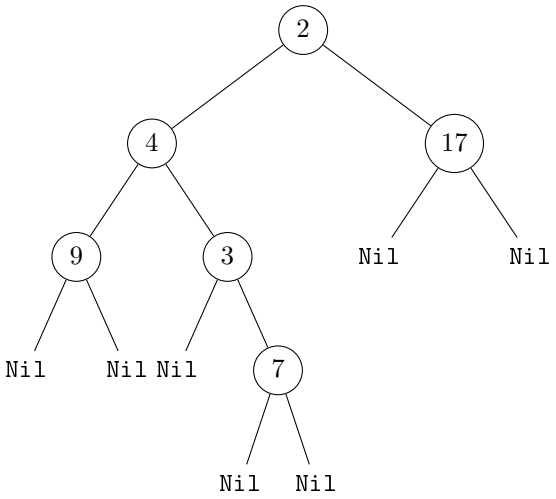
Wir betrachten Operationen auf dem Typ `Tree` der wie folgt definiert ist:

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

Ein Beispielobjekt vom Typ `Tree` ist:

```
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`.

Hinweise:

- Das leere Produkt (= Produkt mit 0 Faktoren) ist 1.

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 (Funktionen höherer Ordnung): (1 + 1 + 1 + 1 + 1 + 1 = 6 Punkte)

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

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

Zwei Beispielobjekte vom Typ `List Int` sind:

```
list :: List Int
list = Cons (-3) (Cons 14 (Cons (-6) (Cons 7 (Cons 1 Nil))))

blist :: List Int
blist = Cons 1 (Cons 1 (Cons 0 (Cons 0 Nil)))
```

Die Liste `list` entspricht also $[-3, 14, -6, 7, 1]$.

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 auftritt. Der Ausdruck `filterList (\x -> x > 10 || x < -5) list` soll dann also zu `(Cons 14 (Cons -6 Nil))` auswerten.

b) Schreiben Sie eine Funktion `divisibleBy :: Int -> List Int -> List Int`, die die Teilliste der Werte zurückgibt, die durch das übergebene Element teilbar sind. Für `divisibleBy 7 list` soll also `Cons 14 (Cons 7 Nil)` zurückgegeben werden. Verwenden Sie dafür `filterList`.

Hinweise:

Die vordefinierte Funktion `mod` berechnet die Modulo-Operation.

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 `plus x y = x + y` der Ausdruck `foldList plus 0 list` zu `-3 + 14 + (-6) + 7 + 1 = 13` ausgewertet werden.

d) Schreiben sie eine Funktion `listMaximum :: List Int -> Int`, die für eine nicht-leere Liste das Maximum berechnet. Verwenden sie hierzu `foldList`. Auf der leeren Liste darf sich Ihre Funktion beliebig verhalten.

Hinweise:

Die vordefinierte Konstante `minBound :: Int` liefert den kleinsten möglichen Wert vom Typ `Int`.

e) Schreiben sie eine Funktion `zipLists :: (a -> b -> c) -> List a -> List b -> List c` die aus zwei Listen eine neue erstellt. Das Element an Position i der resultierenden Liste ist das Ergebnis der Anwendung der übergebenen Funktion auf die beiden Elemente an Position i der Eingabelisten. Falls eine Liste mehr Elemente enthält als die andere, werden die überzähligen Elemente ignoriert. Die Länge der Ausgabeliste ist also gleich der Länge der kürzeren Eingabeliste.

Beispielsweise soll die Anwendung von `zipLists (>) list blist` also `Cons False (Cons True (Cons False (Cons True Nil)))` ergeben.

f) Schreiben sie eine Funktion `skalarprodukt :: List Int -> List Int -> Int`. Diese interpretiert die übergebenen Listen als Vektoren und berechnet das Skalarprodukt. Falls eine Eingabeliste länger ist als die andere, werden die überzähligen Elemente ignoriert. Verwenden sie hierzu `zipLists` und `foldList`.

Für den Aufruf `skalarprodukt blist list` wird also das Ergebnis $1 \cdot (-3) + 1 \cdot 14 + 0 \cdot (-6) + 0 \cdot 7 = 11$ zurückgegeben.

Tutoraufgabe 7 (Unendliche Listen):

a) Implementieren Sie in Haskell die Funktion `odds` vom Typ `[Int]`, welche die Liste aller ungeraden natürlichen Zahlen berechnet.

b) Aus der Vorlesung ist Ihnen die Funktion `primes` bekannt, welche die Liste aller Primzahlen berechnet. Nutzen Sie diese Funktion nun, um die Funktion `primeFactors` vom Typ `Int -> [Int]` in Haskell zu implementieren. Diese Funktion soll zu einer natürlichen Zahl ihre Primfaktorzerlegung als Liste berechnen (auf Zahlen kleiner als 1 darf sich Ihre Funktion beliebig verhalten). Z. B. soll der Aufruf `primeFactors 420` die Liste `[2,2,3,5,7]` berechnen.

Hinweise:

- Die vordefinierten Funktionen `div` und `mod` vom Typ `Int -> Int -> Int`, welche die abgerundete Ganzzahldivision bzw. die Modulo-Operation berechnen, könnten hilfreich sein.

Aufgabe 8 (Unendliche Listen): (6 Punkte)

Geben Sie einen Haskell-Ausdruck an, der zu der aufsteigend sortierten Liste aller *guten Primzahlen* ausgewertet wird. Die n-te Primzahl p_n ist nach einer üblichen Definition genau dann eine gute Primzahl, wenn ihr Quadrat größer als das Produkt der benachbarten Primzahlen ist, d.h. $p_n^2 > p_{n-1} \cdot p_{n+1}$. Die erste gute Primzahl ist daher 5, denn es gilt $5^2 > 3 \cdot 7$.

Hinweise:

- Die vordefinierte Funktion `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]` berechnet eine Liste, in der an der i-ten Position das Ergebnis der Anwendung der übergebenen Funktion auf die beiden Elemente an der i-ten Position der beiden Eingabelisten steht. Sie ist also analog zur Funktion `zipLists` aus Aufgabe 6 e).
- Sie dürfen die Funktion `primes` aus der Vorlesung verwenden.
- Sie können auch die vordefinierte Funktion `drop :: Int -> [a] -> [a]` benutzen. Der Aufruf `drop n xs` gibt die Liste zurück, die man erhält, wenn man die ersten `n` Elemente aus `xs` entfernt.
- Auch die beiden Funktionen `map` und `filter`, die aus der Vorlesung bekannt sind, können hier hilfreich sein.

Tutoraufgabe 9 (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
J. Hensel (jhe)	Assistent
F. Frohn (ffr)	Assistent
D. Korzeniewski (dko)	Assistent
L. Merz (lme)	Tutor
F. Kiunke (fki)	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.

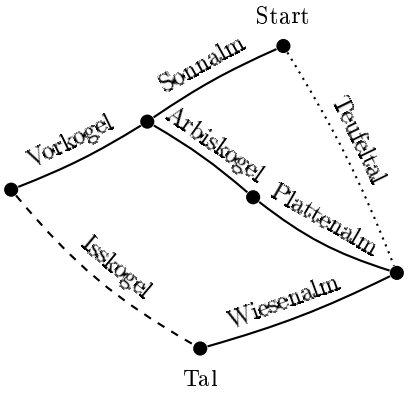
c) Schreiben Sie ein Prädikat `bossVon`, womit Sie abfragen können, wer innerhalb der Übungsbetriebshierarchie einen Rang direkt über dem eines anderen bekleidet. Die Reihenfolge der Ränge ist Professor > Assistent > Tutor > Student. So ist z.B. `bossVon(jhe,lme)` wahr, während `bossVon(dko,fai)` und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

d) Stellen Sie eine Anfrage, mit der Sie alle Personen herausfinden, die in der Übungsbetriebshierarchie direkte Untergebene haben. Dabei sind mehrfache Antworten mit dem gleichen Ergebnis erlaubt.

e) Schreiben Sie schließlich ein Prädikat `vorgesetzt` mit zwei Regeln, 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(jgi, fai)` und `vorgesetzt(dko, fai)` beide wahr, während `vorgesetzt(lme, jhe)` falsch ist.

Aufgabe 10 (Programmieren in Prolog): (2 + 1 + 1.5 + 2.5 = 7 Punkte)

In dieser Aufgabe soll ein Skipisten-Plan in Prolog modelliert und analysiert werden. In der Abbildung sind einfache (blaue) Pisten mit durchgehenden Linien, die einzige mittelschwere (rote) Piste mit einer gestrichelten und die einzige schwere (schwarze) Piste mit einer gepunkteten Linie eingezeichnet.



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

a) Übertragen Sie die oben gegebenen Informationen in eine Wissensbasis für Prolog. Geben Sie hierzu Fakten und/oder Regeln für die Prädikatssymbole `blau`, `rot`, `schwarz`, `start` und `endetIn` an. Hierbei gilt `blau(X)`, falls `X` eine einfache Piste ist, `rot(X)`, falls `X` eine mittelschwere Piste ist, `schwarz(X)`, falls `X` eine schwere Piste ist, `start(X)`, falls die Piste `X` am Ausgangspunkt („Start") beginnt, und `endetIn(X,Y)`, falls die Piste `X` an einem Punkt endet, an dem die Piste `Y` beginnt, oder falls `X` im Tal endet und `Y = tal` gilt.

b) Geben Sie eine Anfrage an das im ersten Aufgabenteil erstellte Programm an, mit der man herausfinden kann, welche Pisten am Startpunkt der Wiesenalm enden.

Hinweise:

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

c) Schreiben Sie ein Prädikat `gleicherStartpunkt`, sodass `gleicherStartpunkt(X,Y)` genau dann gilt, wenn die Pisten `X` und `Y` am gleichen Punkt beginnen. In obiger Abbildung gilt das genau für die Pisten „Sonnalm" und „Teufeltal" bzw. „Vorkogel" und „Arbiskogel". Es ist nicht erlaubt, zur Lösung dieser Teilaufgabe zusätzliche Fakten anzugeben.

d) Schreiben Sie ein Prädikat `anfaengerGeeignet`, sodass `anfaengerGeeignet(X)` genau dann gilt, wenn `X` eine einfache Piste ist und entweder direkt im Tal endet oder in einer anderen Piste, die ihrerseits anfängergeeignet ist. Es ist nicht erlaubt, zur Lösung dieser Teilaufgabe zusätzliche Fakten anzugeben.