Prof. Dr. J. Giesl
D. Cloerkes, S. Dollase, N. Lommen, D. Meier, F. Meyer

Aufgabe 3 (Funktionen höherer Ordnung): (10 + 6 + 4 + 4 + 4 + 7 = 35 Punkte)

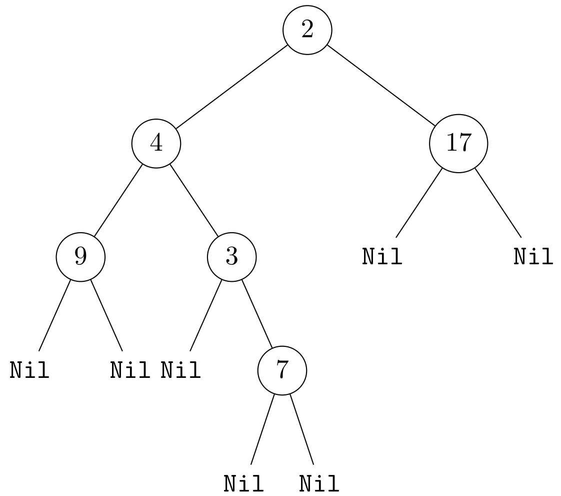
Wir betrachten Operationen auf dem Typ `Tree`, der wie folgt definiert ist:

```
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 mit dem Konstruktor `Node` gebildet 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.

a) Implementieren Sie eine Funktion `foldTree :: (Int -> a -> a -> a) -> a -> Tree -> a`, die dieses allgemeine Muster umsetzt. Bei der Anwendung soll `foldTree` dann alle Vorkommen des Konstruktors `Node` durch die Funktion `f` und alle Vorkommen des Konstruktors `Nil` durch die Konstante `c` ersetzen. Dies ist analog zur Funktion `foldList` in Aufgabe 2(c).

b) Geben Sie unter Nutzung der Funktion `foldTree` alternative Implementierungen für die Funktionen `decTree`, `sumTree` und `flattenTree` an.

c) 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:

- Überlegen Sie, auf welchen Wert `prodTree` den leeren Baum `Nil` abbilden muss, damit die Multiplikation richtig ausgeführt wird.

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

e) Implementieren Sie eine Funktion `mirrorTree`, die einen gespiegelten Baum zurückgibt. Dabei soll die Spiegelung in jedem Knoten vorgenommen werden und nicht nur in der Wurzel. Beim Aufruf von `mirror testTree` soll `Node 2 (Node 17 Nil Nil) (Node 4 (Node 3 (Node 7 Nil Nil) Nil) (Node 9 Nil Nil))` zurückgegeben werden. Verwenden Sie dazu die Funktion `foldTree`.

f) Implementieren Sie eine Funktion `leaves`, die eine Liste mit den Werten der Knoten im Eingabebaum zurückgibt, die Blätter sind, d.h. auf die keine weiteren `Node`-Konstruktoren folgen. Der leere Baum `Nil` hat keine Blätter. Es soll also `leaves testTree = [9,7,17]` gelten. Verwenden Sie dazu die Funktion `foldTree`.

Hinweise:

- Verwenden Sie dazu eine Hilfsfunktion `leavesN :: Int -> [Int] -> [Int] -> [Int]`.
- Ein Knoten `t` ist dann ein Blatt, wenn `leaves` für seine Kinder jeweils den Wert `[]` zurückgibt.

Lösung:

```
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)

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

-- b)
decN = \v l r -> Node (v - 1) l r
sumN = \v l r -> v + l + r
flattenN = \v l r -> v : l ++ r

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

-- c)
prodTree = foldTree (\x y z -> x * y * z) 1

-- d)
incTree = foldTree (\x y z -> Node (x+1) y z) Nil

-- e)
mirrorTree = foldTree (\v l r -> Node v r l) Nil

-- f)
leaves = foldTree leavesN [] where
 leavesN v [] [] = [v]
 leavesN _ xs ys = xs ++ ys
```

Aufgabe 5 (Unendliche Datenstrukturen): (12 + 11 + 12 = 35 Punkte)

In den folgenden Teilaufgaben sollen Sie jeweils einen Haskell-Ausdruck angeben. Wenn Sie dafür eine Funktion schreiben, die eine Eingabe erwartet, so machen Sie deutlich, mit welchen Argumenten die Funktion aufgerufen werden muss, um zur entsprechenden Liste evaluiert zu werden. Sie dürfen die vordefinierten Funktionen `map`, `filter`, `++`, `concat`, `['a'..'z']`, `div`, `mod`, `sum` sowie arithmetische und Vergleichsoperatoren wie `+`, `*`, `==` etc. benutzen. Verwenden Sie keine weiteren vordefinierten Funktionen, wenn sie nicht explizit erwähnt sind.

a) Geben Sie einen Haskell-Ausdruck an, der zu einer unendlichen Liste aller Palindrome ausgewertet wird. Ein Palindrom ist ein `String`, der vorwärts und rückwärts gelesen gleich ist. Somit ist `"anna"` ein Beispiel für ein Palindrom. Wir betrachten in dieser Aufgabe ausschließlich `String`s, die aus den Zeichen `'a'` bis `'z'` bestehen. Die berechnete Liste soll bezüglich der Länge ihrer Elemente aufsteigend sortiert sein.

Sie dürfen die folgende Hilfsfunktion `strings` benutzen. Diese berechnet alle `Strings` der Länge `n`, wobei `n` das erste Argument der Funktion ist.

```
strings :: Int -> [String]
strings 0 = [""]
strings n = concat (map (\x -> map (\tail -> x:tail) tails) ['a'..'z'])
  where tails = strings (n-1)
```

Hinweise:

- Die Funktion `reverse :: [a] -> [a]` dreht eine Liste um und `concat :: [[a]] -> [a]` hängt alle Elementlisten hintereinander, d.h. `concat [[1,2],[3]]` ergibt `[1,2,3]`.

b) Geben Sie einen Haskell-Ausdruck an, der zu der aufsteigend sortierten Liste aller *semiperfekten Zahlen* ausgewertet wird. Eine Zahl $x \geq 2$ ist genau dann semiperfekt, wenn die Summe *aller oder einiger* ihrer echten Teiler gleich x ist. Betrachten Sie als Beispiel die Zahl 12: Ihre echten Teiler sind $1, 2, 3, 4$ und 6 und es gilt $2 + 4 + 6 = 12$, also ist 12 eine semiperfekte Zahl.

Hinweise:

- Die Funktion `any :: (a -> Bool) -> [a] -> Bool` testet, ob ein Element einer Liste das als erstes Argument übergebene Prädikat erfüllt.
- Die Funktion `subsequences :: [a] -> [[a]]` berechnet alle Teillisten der als Argument übergebenen Liste. Es gilt zum Beispiel:

  `subsequences [1,2,3] = [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]`

  Damit Sie die Funktion `subsequences` nutzen können, muss die erste Zeile der Datei mit Ihrer Lösung “`import Data.List`” lauten.
- Sie dürfen die Funktionen `divisors` und `perfect` aus Aufgabe 4(b) verwenden.

c) Geben Sie einen Haskell-Ausdruck an, der zu der aufsteigend sortierten Liste aller Fibonacci-Zahlen evaluiert wird. Die Fibonacci-Zahlen haben Sie bereits auf Blatt 9 kennengelernt. Greifen Sie dafür *nicht* auf einen Ausdruck zurück, der die n-te Fibonacci-Zahl berechnet.

Hinweise:

- Überlegen Sie, wie Sie die Effizienzüberlegungen von Blatt 9 auch in dieser Aufgabe umsetzen können. Für eine ineffiziente Lösung, bei der Elemente der Liste mehrfach evaluiert werden, werden keine Punkte vergeben.
- Es bietet sich an, die Hilfsfunktion `fibInit :: Int -> Int -> [Int]` zu implementieren, die die unendliche Liste der Fibonacci-Zahlen mit beliebigen Initialwerten berechnet, vgl. hierzu Aufgabe 8 auf Blatt 9.

Lösung:

```
import Data.List

strings :: Int -> [String]
strings 0 = [""]
strings n =  concat (map (\x -> map (\tail -> x:tail) tails) ['a'..'z'])
  where tails = strings (n-1)

palindrom :: [String]
palindrom = palindrom' 0
  where palindrom' n = (filter (\x -> x == reverse x) (strings n)) ++ palindrom' (n+1)

--palindrom' stellt alternative Loesungsmoeglichkeit dar
palindrom' :: [String]
palindrom' =
  simplePalindroms ++ concat (map (\s -> map (\c -> c:s ++ [c]) chars) palindrom')
  where chars = ['a'..'z']
        simplePalindroms = "" : map (\c -> [c]) chars
```

```
divisors :: Int -> [Int]
divisors x = filter (\y -> mod x y == 0) [1..div x 2]

semiperfect :: [Int]
semiperfect = filter (\x -> any (\xs -> sum xs == x) (subsequences (divisors x))) [2..]

fib :: [Int]
fib = 0 : 1 : fibInit 0 1

fibInit :: Int -> Int -> [Int]
fibInit n m = (n+m) : fibInit m (n+m)
```

Aufgabe 8 (Programmieren in Prolog): (3 + 4 + 6 + 4 + 5 + 8 = 30 Punkte)

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
D. Cloerkes (dcl)	Assistent
S. Dollase (sdo)	Assistent
N. Lommen (nlo)	Assistent
D. Meier (dme)	Assistent
F. Meyer (fme)	Assistent
J. Drew (jdr)	Hiwi
F. Rupprath (fru)	Hiwi
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.
Nutzen Sie bei Ihrer Implementierung jeweils Prädikate aus den vorangegangenen Aufgabenteilen. Benutzen Sie **keine vordefinierten Prädikate**. Achten Sie auf die **korrekte Schreibweise** der Namen aus der Aufgabenstellung. Achten Sie bei Ihrer Implementierung darauf, dass diese allgemein sein soll und *nicht* nur für das angegebene Beispiel funktionieren soll. Es nutzt also nichts, in weiteren Fakten konkrete Fälle aus dem Beispiel zu sammeln.

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 > Hiwi > Student. So ist z.B. `bossVon(sdo,jdr)` wahr, während `bossVon(jgi,fai)` und `bossVon(ner,mus)` 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 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(dme, fme)` wahr, während `hatGleichenRang(jgi, mus)` 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, 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 > Hiwi > Student gilt). So sind z.B. `vorgesetzt(jgi, fai)` und `vorgesetzt(fru, fai)` beide wahr, während `vorgesetzt(mus, fme)` falsch ist.

Lösung:

```
% Teilaufgabe a)
person(jgi).
person(dcl).
person(sdo).
person(nlo).
person(dme).
person(fme).
person(jdr).
person(fru).
person(fai).
person(ner).
person(mus).

hatRang(jgi, professor).
hatRang(dcl, assistent).
hatRang(sdo, assistent).
hatRang(nlo, assistent).
hatRang(dme, assistent).
hatRang(fme, assistent).
hatRang(jdr, hiwi).
hatRang(fru, hiwi).
hatRang(fai, student).
hatRang(ner, student).
hatRang(mus, student).

% Teilaufgabe b)
% ?- hatRang(X, assistent).
% Ausgabe:
% X = dcl ;
% X = sdo ;
% X = nlo ;
% X = dme ;
% X = fme.

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

% Teilaufgabe d)
% ?- bossVon(X, _).
% Ausgabe:
```

```
% X = jgi ;
% X = jgi ;
% X = jgi ;
% X = jgi ;
% X = jgi ;
% X = dcl ;
% X = dcl ;
% X = sdo ;
% X = sdo ;
% X = nlo ;
% X = nlo ;
% X = dme ;
% X = dme ;
% X = fme ;
% X = fme ;
% X = jdr ;
% X = jdr ;
% X = jdr ;
% X = fru ;
% X = fru ;
% X = fru ;
% false.

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

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