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

Tutoraufgabe 1 (Überblickswissen):

a) Inwiefern unterscheiden sich zyklische Listen in Java und Haskell konzeptuell?

b) In Prolog gibt es zwei Arten von Klauseln: Fakten und Regeln. Warum sind die Fakten eigentlich nur syntaktischer Zucker?

Lösung:

a) In Java gibt es Objekte, und in einer zyklischen Liste der Länge n kommt man nach n-facher Anwendung des Listennachfolgers wieder beim selben Objekt an, und nicht nur bei dem gleichen Objekt. In Haskell dagegen gibt es keinen Unterschied zwischen demselben und dem gleichen Objekt (d.h. einem möglicherweise anderen Objekt mit identischen Attributwerten). Man kann somit nicht feststellen, ob sich der Kreis in einer zyklischen Haskell-Liste wieder schließt, oder ob man nicht doch nur auf halbem Weg bei einem Element angekommen ist, dass (zufällig) gleich dem Ursprungselement ist. In Haskell wird also lediglich nach einer zyklischen Vorschrift eine in Wirklichkeit unendliche Liste erzeugt, während in Java im Speicher tatsächlich eine zyklische Liste vorliegt, bei der die n enthaltenen Elemente einander zyklisch referenzieren.

b) Man kann jedes Faktum `fact(arg1,...,argN).` durch die Regel `fact(arg1,...,argN) :- true.` ersetzen, wodurch sich die Semantik des Programms nicht ändert. Statt also die Fakten an sich als wahr zu definieren, kann man sie auch in einem weiteren Schritt zum atomaren Wahrheitswert `true` ableiten. Das läuft am Ende auf dasselbe hinaus, liest sich aber nicht ganz so intuitiv, weshalb eine eigene Schreibweise für Fakten eingeführt wurde.

Tutoraufgabe 2 (Funktionen höherer Ordnung):

Wir betrachten Funktionen auf dem parametrisierten Typ `List a`, der 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 $[-3, 14, -6, 7, 1]$.
Sie dürfen vordefinierte arithmetische Operatoren wie `+`, `*` etc. und Relationen wie `==` benutzen. Verwenden Sie keine weiteren vordefinierten Funktionen, wenn sie nicht explizit erwähnt sind.

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`, wobei `divisibleBy x xs` die Teilliste der Werte der Liste `xs` zurückgibt, die durch `x` teilbar sind. Für `divisibleBy 7 list` soll also `Cons 14 (Cons 7 Nil)` zurückgegeben werden. Verwenden Sie dafür `filterList`.

Hinweise:

Sie dürfen die vordefinierte Funktion `mod x y` verwenden, die die Modulo-Operation berechnet.

c) Schreiben Sie eine Funktion `foldList :: (a -> b -> b) -> b -> List a -> b`, die 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:

Sie dürfen die vordefinierte Konstante `minBound :: Int` benutzen, die den kleinsten möglichen Wert vom Typ `Int` liefert.

e) Schreiben Sie eine Funktion `mapList :: (a -> b) -> List a -> List b`, die sich auf unseren selbstdefinierten Listen wie `map` auf den vordefinierten Listen verhält. Es soll also die als erster Parameter übergebene Funktion auf jedes Element angewandt werden, um die Ausgabeliste zu erzeugen. Der Ausdruck `mapList (\x -> 2*x) list` soll dann also zu `Cons (-6) (Cons 28 (Cons (-12) (Cons 14 (Cons 2 Nil))))` auswerten.

Verwenden Sie hierzu neben der Typdeklaration nur eine weitere Zeile, in der Sie `mapList` mittels `foldList` definieren.

f) **(Video)** 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.

g) **(Video)** 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.

Hinweise:

Infix-Operatoren wie `+` lassen sich auch als Funktionen in der normalen Präfix-Schreibweise verwenden, indem man sie in Klammern setzt. So steht `(+) 2 3` also für `2+3`.

Lösung:

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

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 (Cons 1 Nil))))

-- a)
filterList :: (a -> Bool) -> List a -> List a
filterList _ Nil = Nil
filterList f (Cons x xs) | f x  = Cons x rest
```

```
                          | otherwise = rest
  where rest = filterList f xs

-- b)
divisibleBy :: Int -> List Int -> List Int
divisibleBy n = filterList (\x -> (mod x n) == 0)

-- c)
foldList :: (a -> b -> b) -> b -> List a -> b
foldList _ c Nil          = c
foldList f c (Cons x xs) = f x (foldList f c xs)

-- d)
listMaximum :: List Int -> Int
listMaximum = foldList max minBound

-- e)
mapList :: (a -> b) -> List a -> List b
mapList f xs = foldList (\x y -> Cons (f x) y) Nil xs

-- f)
zipLists :: (a -> b -> c) -> List a -> List b -> List c
zipLists _ Nil _ = Nil
zipLists _ _ Nil = Nil
zipLists f (Cons x xs) (Cons y ys) = Cons (f x y)  (zipLists f xs ys)

-- g)
skalarprodukt :: List Int -> List Int -> Int
skalarprodukt x y = foldList (+) 0 (zipLists (*) x y)
```

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

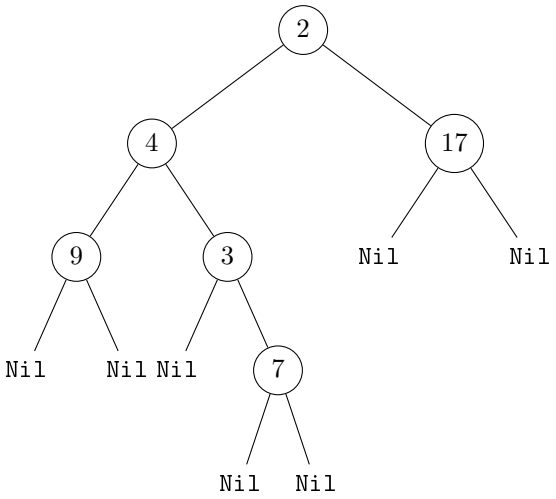
Wir betrachten Operationen auf dem Typ `Tree`, der wie folgt definiert ist:

```
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
```

Tutoraufgabe 4 (Unendliche Datenstrukturen):

a) Implementieren Sie in Haskell die Funktion `evenlist` vom Typ `[Int]`, welche die Liste aller geraden natürlichen Zahlen (mit 0) berechnet.

b) Geben Sie einen Haskell-Ausdruck an, der zu der aufsteigend sortierten Liste aller *perfekten Zahlen* ausgewertet wird. Eine Zahl $x \geq 2$ ist genau dann perfekt, wenn die Summe ihrer echten Teiler gleich x ist, vgl. Aufgabe 7(c) auf Übungsblatt 9. Betrachten Sie als Beispiel die Zahl 6: Ihre echten Teiler sind $1, 2$ und 3 und es gilt $1 + 2 + 3 = 6$, also ist 6 eine perfekte Zahl.

Sie dürfen die folgende Hilfsfunktion `divisors` benutzen. Diese berechnet alle echten Teiler der als Argument übergebenen Zahl.

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

Hinweise:

- Die Funktion `sum :: [Int] -> Int` berechnet die Summe aller Elemente einer Liste.
- Für jede Zahl x erzeugt `[x..]` die unendliche Liste `[x,x+1,x+2,...]`, und für $x \leq y$ erzeugt `[x..y]` die Liste `[x,x+1,...,y]`.

c) 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 2 darf sich Ihre Funktion beliebig verhalten). Beispielsweise 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.

Lösung:

a)
```
evenlist :: [Int]
evenlist = 0 : map (+ 2) evenlist
```

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

perfect :: [Int]
perfect = filter (\x -> x == sum (divisors x)) [2..]
```

c)
```
primeFactors :: Int -> [Int]
primeFactors = pHelper primes
   where
   pHelper (x:xs) y | y == 1 = []
                    | mod y x == 0 = x : pHelper (x:xs) (div y x)
                    | otherwise = pHelper xs y
```

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 `Strings`, 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:

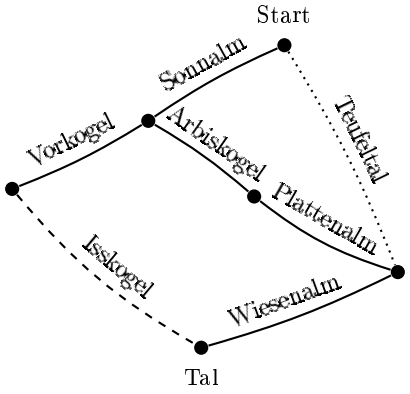
- Überlegen Sie, wie Sie die Effizienzüberlegungen von Blatt 9 auch in dieser Aufgabe umsetzen können. Für eine ineffiziente Lösung, bei der Elemente der Liste mehrfach evaluiert werden, werden keine Punkte vergeben.
- Es bietet sich an, die Hilfsfunktion `fibInit :: Int -> Int -> [Int]` zu implementieren, 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)
```

Tutoraufgabe 6 (Programmieren in Prolog):

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.



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 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 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“.

d) Schreiben Sie ein Prädikat `erreichbar`, sodass `erreichbar(X,Y)` genau dann gilt, wenn es einen Weg von der Piste `X` in Richtung Tal gibt, der über die Piste `Y` führt.

e) Schreiben Sie ein Prädikat `moeglicheSchlusspiste`, sodass `moeglicheSchlusspiste(X,S)` genau dann gilt, wenn es es einen Weg von der Piste `X` ins Tal gibt, der als Letztes über die Piste `S` führt.

f) Schreiben Sie ein Prädikat `treffpisten`, sodass `treffpisten(X,Y,T)` genau dann gilt, wenn die Piste `T` sowohl auf einem Weg von der Piste `X` in Richtung Tal als auch auf einem Weg von der Piste `Y` in Richtung Tal liegt.

Hinweise:

- Das Tal ist keine Piste. Um diesen Fall abzufangen, können Sie mit dem zweistelligen Prädikat `\=` auf Ungleichheit prüfen.

g) Schreiben Sie ein Prädikat `anfaengerGeeignet`, sodass `anfaengerGeeignet(X)` genau dann gilt, wenn es einen Weg von `X` in Richtung Tal gibt, auf dem nur einfache Piste liegen, einschließlich der Piste `X`.

Lösung:

```
% Teilaufgabe a)
blau(sonnalm).
blau(vorkogel).
blau(arbiskogel).
blau(plattenalm).
blau(wiesenalm).
rot(isskogel).
schwarz(teufeltal).

start(sonnalm).
start(teufeltal).

endetIn(sonnalm, vorkogel).
endetIn(sonnalm, arbiskogel).
endetIn(vorkogel, isskogel).
endetIn(arbiskogel, plattenalm).
endetIn(plattenalm, wiesenalm).
endetIn(teufeltal, wiesenalm).
endetIn(isskogel, tal).
endetIn(wiesenalm, tal).

% Teilaufgabe b)
% ?- endetIn(X, wiesenalm).
% Ausgabe:
% X = plattenalm ;
% X = teufeltal.

% Teilaufgabe c)
gleicherStartpunkt(X, Y) :- start(X), start(Y).
gleicherStartpunkt(X, Y) :- endetIn(Z, X), endetIn(Z, Y).

% Teilaufgabe d)
erreichbar(X,Y) :- endetIn(X,Y).
erreichbar(X,Z) :- endetIn(X,Y), erreichbar(Y,Z).

% Teilaufgabe e)
moeglicheSchlusspiste(S,S) :- endetIn(S,tal).
moeglicheSchlusspiste(X,S) :- erreichbar(X,S), endetIn(S,tal).

% Teilaufgabe f)
treffpisten(X,Y,T) :- erreichbar(X,T), erreichbar(Y,T), T \= tal.

% Teilaufgabe g)
anfaengerGeeignet(X) :- blau(X), endetIn(X, tal).
anfaengerGeeignet(X) :- blau(X), endetIn(X, Y), anfaengerGeeignet(Y).
```

Tutoraufgabe 7 (Programmieren in Prolog (Video)):

In dieser Aufgabe soll ein Bewertungssystem in Prolog modelliert und analysiert werden.
Jede Unterkunft kann mit 1 bis 5 Sternen bewertet werden.

Unterkunft	Bewertung
Waldhotel (wh)	4 Sterne
Berghotel (bh)	5 Sterne
Hotel am See (sh)	2 Sterne
Pilgerhotel (ph)	1 Sterne
Jugendherberge (jh)	3 Sterne
Bett bei Freunden (fb)	5 Sterne

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 oben gegebenen Informationen in eine Wissensbasis für Prolog. Geben Sie hierzu Fakten und/oder Regeln für die Prädikatssymbole `istUnterkunft`, `hatEinenStern`, `hatZweiSterne`, `hatDreiSterne`, `hatVierSterne` und `hatFuenfSterne` an. Hierbei gilt `istUnterkunft(X)`, falls `X` eine Unterkunft ist, `hatEinenStern(X)`, falls `X` eine mit einem Stern bewertete Unterkunft ist, `hatZweiSterne(X)`, falls `X` eine mit zwei Sternen bewertete Unterkunft ist, usw.

b) Geben Sie eine Anfrage an das im ersten Aufgabenteil erstellte Programm an, mit der man herausfinden kann, welche Unterkünfte mit fünf Sternen bewertet worden sind.

Hinweise:

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

c) Schreiben Sie ein Prädikat `hatEinenSternWeniger`, womit Sie abfragen können, ob eine Unterkunft mit genau einem Stern weniger bewertet wurde als eine zweite Unterkunft. So ist beispielsweise `hatEinenSternWeniger(wh, fb)` wahr, während `hatEinenSternWeniger(sh, bh)` und `hatEinenSternWeniger(jh, sh)` beide falsch sind.

d) Stellen Sie eine Anfrage, mit der Sie (bei wiederholter Eingabe von „`;`“) alle Unterkünfte herausfinden, die so bewertet wurden, dass es mindestens eine weitere Unterkunft gibt, welche mit genau einem Stern weniger bewertet worden ist. Dabei sind mehrfache Antworten mit dem gleichen Ergebnis erlaubt.

e) Schreiben Sie ein Prädikat `hatWenigerSterne`, mit welchem Sie Paare von Unterkünften abfragen können, bei denen die erste Unterkunft schlechter bewertet worden ist als die zweite Unterkunft. So ist z.B. `hatWenigerSterne(ph, bh)` wahr, während `hatWenigerSterne(bh, fb)` und `hatWenigerSterne(jh, ph)` beide falsch sind.

Lösung:

```
% Teilaufgabe a)
istUnterkunft(wh).
istUnterkunft(bh).
istUnterkunft(sh).
istUnterkunft(ph).
istUnterkunft(jh).
istUnterkunft(fb).

hatEinenStern(ph).
hatZweiSterne(sh).
hatDreiSterne(jh).
hatVierSterne(wh).
hatFuenfSterne(bh).
hatFuenfSterne(fb).
```

```
% Teilaufgabe b)
% ?- hatFuenfSterne(X).
% Ausgabe:
% X = bh ;
% X = fb.

% Teilaufgabe c)
hatEinenSternWeniger(X, Y) :- hatEinenStern(X), hatZweiSterne(Y).
hatEinenSternWeniger(X, Y) :- hatZweiSterne(X), hatDreiSterne(Y).
hatEinenSternWeniger(X, Y) :- hatDreiSterne(X), hatVierSterne(Y).
hatEinenSternWeniger(X, Y) :- hatVierSterne(X), hatFuenfSterne(Y).

% Teilaufgabe d)
% ?- hatEinenSternWeniger(_, Y).
% Ausgabe:
% Y = sh ;
% Y = jh ;
% Y = wh ;
% Y = bh ;
% Y = fb.

% Teilaufgabe e)
hatWenigerSterne(X, Y) :- hatEinenSternWeniger(X, Y).
hatWenigerSterne(X, Y) :- hatEinenSternWeniger(X, Z), hatWenigerSterne(Z, Y).
```

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).
```