1. Datentypen

i).

Geben Sie für die folgenden Typdeklarationen die Namen und Typen der automatisch generierten Konstruktoren und Selektoren an:

```
data Ampelfarbe = Rot | Gelb | Gruen
data Menge = Stueck Integer | Kilogramm Double | Liter Double
data HausOderPostfach = Haus {
    strasse :: String,
    hausNummer :: Integer
  }
  | Postfach {
    postfachNummer :: Integer
  }
data Adresse = Adresse {
    hausOderPostfach :: HausOderPostfach,
    ort :: String,
    plz :: Integer
  }
```

ii).

Geben Sie für den Datentyp Menge eine äquivalente Deklaration an, bei der automatisch Selektorfunktionen generiert werden. Wählen Sie sinnvolle Namen für die Selektorfunktionen.

iii).

Geben Sie für die Datentypen HausOderPostfach und Adresse äquivalente Deklarationen an, bei denen nicht automatisch Selektoren generiert werden.

iv).

Programmieren Sie aufbauend auf den Typdeklarationen aus Teilaufgabe 1 Selektorfunktionen mit den folgenden Signaturen:

```
strasseVonAdresse :: Adresse -> String
postfachNummerVonAdresse :: Adresse -> Integer
```

2. Pattern-Matching

Gegeben sind die folgenden Typdeklarationen:

```
data SI = SI String Integer
data SIListe = Leer | NichtLeer SI SIListe
data VielleichtSI = Nicht | Doch SI
```

i).

Bestimmen Sie die Typen der Variablen in den folgenden Mustern. Begründen Sie Ihre Antworten kurz.

```
loesche :: SI -> SIListe -> SIListe
loesche a Leer = error "nicht da"
loesche (SI b c) (NichtLeer (SI d e) f )
               | b == d && c == e = f
               | otherwise        = NichtLeer (SI d e) (loesche (SI b c) f )

ersetze :: SI -> SIListe -> SIListe
ersetze g Leer = Leer
ersetze h@(SI i _) (NichtLeer j@(SI k _) l) =
               if i == k then NichtLeer h (ersetze h l)
                         else NichtLeer j (ersetze h l)

findeAnIdx :: Integer -> SIListe -> VielleichtSI
findeAnIdx _ Leer                        = Nicht
findeAnIdx m (NichtLeer n o) | m == 0    = Doch n
                             | otherwise = findeAnIdx ( pred m) o
```

Hinweis:
Das @ bindet den „zerlegten“ Datentyp an den vorstehenden Bezeichner, damit dieser nicht wieder zusammen gesetzt werden muss.

ii).

Gegeben sind folgende Definitionen:

```
liste1 = NichtLeer (SI "Bert" 7)
         (NichtLeer (SI "Bianca" 9)
         (NichtLeer (SI "Bert" 7) Leer))
liste2 = NichtLeer (SI "Bert" 8)
         (NichtLeer (SI "Robert" 7) Leer)
liste3 = NichtLeer (SI "Robert" 8)
         (NichtLeer (SI "Robert" 7) Leer)
```

Geben Sie die Resultate folgender Funktionsanwendungen an. Begründen Sie Ihre Antworten kurz.
a)

```
loesche (SI "Bert" 7) liste1
```

b)

```
ersetze (SI "Robert" 9) liste2
```

c)

```
ersetze (SI "Robert" 9) liste3
```

d)

```
findeAnIdx 1 liste3
```

3. Arithmetische Ausdrücke

Wir betrachten einen rekursiven Datentyp `Ausdruck`, dessen Werte arithmetische Ausdrücke darstellen. Als Operationen sind Addition, Subtraktion und Multiplikation zugelassen, atomare Einheiten können Konstanten und Variablen sein. Die Deklaration von `Ausdruck` ist wie folgt:

```
data Ausdruck = Konstante Integer
  | Variable String
  | Summe Ausdruck Ausdruck
  | Differenz Ausdruck Ausdruck
  | Produkt Ausdruck Ausdruck
  deriving (Show) -- ermoeglicht Ausgabe der Werte im GHCi
```

Der Ausdruck $x + (3 - x) \cdot y$ wird beispielsweise repräsentiert durch:

```
Summe (Variable "x") (Produkt (Differenz (Konstante 3) (Variable "x"))
  (Variable "y"))
```

i).

Geben Sie die Repräsentationen folgender arithmetischer Ausdrücke an:
a) $-x$
b) $3 \cdot x + y$
c) $3 \cdot (x + y)$
d) $3 + x \cdot y$

ii).

Geben Sie die arithmetischen Ausdrücke an, die folgendermaßen repräsentiert werden:
a)

```
Produkt (Differenz (Variable "x") (Konstante 0)) (Variable "y")
```

b)

```
Differenz (Variable "x") (Produkt (Konstante 0) (Variable "y"))
```

c)

```
Summe (Produkt (Konstante 3) (Variable "z")) (Konstante 5)
```

iii).

Programmieren Sie eine Haskell-Funktion `ausdruckNachString::Ausdruck -> String` welche arithmetische Ausdrücke in Zeichenfolgen konvertiert. Die ausgegebenen Zeichenfolgen sollen die arithmetischen Ausdrücke in Haskell-Syntax darstellen. Es ist z.B. folgendes möglich:

```
ausdruckNachString (Differenz (Konstante 3) (Variable "x")) = "3␣-␣x"
```

Hinweis: Zum Aneinanderhängen von Zeichenfolgen kann der Operator `++` verwendet werden.

```
(++) :: String -> String -> String
```

Ganzzahlen können durch Anwenden der Funktion `show` in ihre Zifferndarstellung konvertiert werden. Es ergibt sich beispielsweise

```
show 325 = "325"
```

4. Rekursion über Listen

Die folgenden Teilaufgaben sollen ohne Verwendung vordefinierter Bibliotheksfunktionen gelöst werden. Berücksichtigen Sie, dass die zu programmierenden Funktionen nicht für alle Eingaben sinnvoll definiert sind. In diesen Fällen sollen entsprechende Fehlermeldungen ausgegeben werden.

i).

Programmieren Sie eine Haskell-Funktion `elementAnIdx :: Integer -> [el] -> el`, die zu einem Index `i` und einer Liste das i-te Element dieser Liste bestimmt. Das erste Element einer nichtleeren Liste habe dabei den Index 0. Es ergibt sich zum Beispiel:

```
elementAnIdx 2 [3, 4, 5, 6] = 5
```

ii).

Programmieren Sie eine Haskell-Funktion `snoc :: [el] -> el -> [el]`, die an eine Liste ein weiteres Element anhängt. Es ergibt sich zum Beispiel:

```
snoc [1, 2, 3] 5 = [1, 2, 3, 5]
```

iii).

Programmieren Sie eine Haskell-Funktion `rueckwaerts :: [el] -> [el]`, welche eine Liste umdreht, die Elemente der Liste also in umgekehrter Reihenfolge zurück gibt.

iv).

Programmieren Sie eine Haskell-Funktion `praefix :: Integer -> [el] -> [el]`, die für eine gegebene Zahl n und eine Liste l die Liste der ersten n Elemente von l zurück gibt.

v).

Programmieren Sie eine Haskell-Funktion `suffix :: Integer -> [el] -> [el]`, die für eine Zahl n und eine Liste l die Liste der letzten n Elemente von l zurückgibt.

5. Mengen als Listen

Mengen können in Haskell durch Listen repräsentiert werden. Allerdings unterscheiden sich Mengen und Listen in zwei Punkten:

- Die Elemente einer Liste besitzen eine Reihenfolge, die Elemente von Mengen nicht.
- Derselbe Wert darf in einer Liste mehrfach vorkommen, in einer Menge nicht.

Zur Darstellung von Mengen verwenden wir daher nur solche Listen, deren Elemente streng monoton wachsen. Die Reihenfolge der Elemente ist damit festgelegt und eine Dopplung von Werten ist nicht möglich.

Es sollen nun Mengenoperationen auf Basis von Listen implementiert werden. Geben Sie für folgende Funktionen deren allgemeinste Typen an. Implementieren Sie anschließend die Funktionen. Sie können davon ausgehen, dass als Argumente übergebene Mengendarstellungen immer die oben genannte Eigenschaft erfüllen. Sie müssen aber sicher stellen, dass Sie keine ungültigen Mengendarstellungen als Resultate generieren.

i).

Die Funktion `istElement` entscheidet, ob eine Wert in einer Menge enthalten ist und implementiert damit die Relation 2. Es ergibt sich z.B.

```
4 `istElement` [1, 3, 5, 7]   ⇝ False
5 `istElement` [1, 3, 5, 7]   ⇝ True
```

ii).

Die Funktion `istTeilmenge` entscheidet, ob eine Menge Teilmenge einer anderen Menge ist und implementiert damit die Relation $\subseteq$. Es ergibt sich z.B.

```
[1, 3] `istTeilmenge` [1, 2, 3, 4, 5]      ⇝ True
[1, 3, 6] `istTeilmenge` [1, 2, 3, 4, 5]   ⇝ False
```

iii).

Die Funktion `vereinigung` bestimmt die Vereinigungsmenge zweier Mengen und implementiert damit die Funktion $\cup$. Es ergibt sich z.B.

```
[1, 2, 4] `vereinigung` [3, 4, 5] ⇝ [1, 2, 3, 4, 5]
```

iv).

Die Funktion `schnitt` bestimmt die Schnittmenge zweier Mengen und implementiert damit die Funktion $\cap$. Es ergibt sich z.B.

```
[1, 2, 4] `schnitt` [3, 4, 5] ⇝ [4]
```

6. Funktionen als Listen

Wir stellen Funktionen als Listen von Paaren dar. Die Liste zu einer Funktion f enthält für jedes x, für das $f(x)$ definiert ist, das Paar $(x, f(x))$. Die Paare sind innerhalb der Liste aufsteigend nach den Funktionsargumenten sortiert. Kein Paar kommt mehrfach vor.

i).

Programmieren Sie eine Haskell-Funktion `resultat`, welche zu einer als Liste dargestellten Funktion f und einem Wert x des Definitionsbereichs den entsprechenden Funktionswert $f(x)$ liefert. Ist $f(x)$ nicht definiert, soll *resultat* mit einer Fehlermeldung abbrechen. Es ergibt sich z.B.

```
resultat [(-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4)] 2 ⇝ 4
```

ii).

Häufig ist es nicht erwünscht, dass Funktionen mit Fehlermeldungen abbrechen. Stattdessen werden oft Werte des Datentyps `Maybe` als Funktionsresultate verwendet. `Maybe` ist im Prelude wie folgt definiert:

```
data Maybe wert = Nothing | Just wert
```

Der Wert `Nothing` signalisiert einen Fehler, während ein Wert `Just x` die erfolgreiche Berechnung des Wertes x darstellt. Modifizieren Sie die Funktion `resultat` zu einer Funktion `evtlResultat`, welche nicht mit Fehlermeldungen abbricht, sondern `Maybe`- Werte liefert. Es ergibt sich z.B.

```
evtlResultat [(-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4)] 3   ⇝ Nothing
evtlResultat [(-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4)] 2   ⇝ Just 4
```

iii).

Programmieren Sie eine Haskell-Funktion `urbilder`, die zu einer gegebenen Funktion f und einem gegebenen Wert y die Menge $\{x|f(x) = y\}$ berechnet. Diese Menge soll wie in Aufgabe 1 beschrieben dargestellt werden. Es ergibt sich z.B.

```
urbilder [(-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4)] 4 ⇝ [-2, 2]
```

iv).

Programmieren sie eine Haskell-Funktion `echteArgumente`, welche zu einer gegebenen Funktion f die Menge aller x liefert, für die es einen Funktionswert $f(x)$ gibt. Verwenden Sie wieder die Mengendarstellung aus Aufgabe 1. Es ergibt sich z.B.

```
echteArgumente [(-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4)] ⇝ [-2,-1, 0, 1, 2]
```

v).

Wir betrachten nun Funktionen, deren Definitions- und Wertebereich identisch sind. Ein Wert x ist Fixpunkt einer solchen Funktion f , wenn $f(x) = x$ gilt. Programmieren Sie eine Haskell-Funktion `fixpunkte`, die zu einer gegebenen Funktion die Menge aller Fixpunkte berechnet. Diese Menge soll wieder wie in Aufgabe 1 beschrieben dargestellt werden. Es ergibt sich z.B.

```
fixpunkte [(-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4)] ⇝ [0, 1]
```

vi).

Programmieren Sie eine Haskell-Funktion `funKomposition`, die aus zwei Funktionen f und g deren Komposition $f \circ g$ berechnet. Es ergibt sich z.B.

```
[(1, 0), (2, 1)] `funKomposition` [(-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4)] ⇝ [(-1, 0), (1, 0)]
```

7. Relationen als Listen

Eine Relation ist eine Menge von Paaren. Wir wollen solche Mengen von Paaren in Haskell als Listen von Paaren darstellen. Wir wollen allerdings keine besonderen Anforderungen an diese Listen stellen. Paare können in einer Liste in beliebiger Reihenfolge stehen und dürfen auch mehrfach auftreten.

Die Funktionskomposition $\circ$ kann zu einer Relationskomposition verallgemeinert werden. Es seien $R_1 \subseteq A \times B$ und $R_2 \subseteq B \times C$ zwei Relationen. Die Komposition $R_2 \circ R_1 \subseteq A \times C$ ist wie folgt definiert:

$$R_2 \circ R_1 = \{(a, c) \mid \exists\, b : (a, b) \in R_1 \wedge (b, c) \in R_2\}$$

Definieren Sie eine Funktion `relKomposition`, die die Relationskomposition umsetzt. Verwenden Sie dafür eine List-Comprehension. Es ergibt sich z.B.

```
[(4,-2), (1,-1), (0, 0), (1, 1), (4, 2)] `relKomposition` [(0, 1), (1, 2)] ⇝ [(0,-1), (0, 1)]
```

8. Instanziierung von Typklassen

Wir betrachten den im letzten Aufgabenblatt vorgestellten Typ `Ausdruck` für arithmetische Ausdrücke. Dieser ist wie folgt definiert:

```
data Ausdruck = Konstante Integer
  | Variable String
  | Summe Ausdruck Ausdruck
  | Differenz Ausdruck Ausdruck
  | Produkt Ausdruck Ausdruck
```

Beachten Sie, dass wir diesmal die Klausel `deriving (Show)` weggelassen haben.

i).

Machen Sie `Ausdruck` zu einer Instanz von `Eq`. Zwei Ausdrücke sollen nur dann als gleich gelten, wenn sie strukturgleich sind. Die Ausdrücke x und $x + 0$ gelten z.B. als ungleich.

ii).

Machen Sie `Ausdruck` zu einer Instanz von `Show`. Es genügt dabei, die Funktion `show` zu implementieren, welche Ausdrücke in Zeichenketten umwandelt. Ihre `show`- Implementierung soll die gleichen Resultate liefern, wie die `show`-Implementierung, welche durch `deriving(Show)` automatisch erzeugt wird. Es ergibt sich z.B.

```
show (Differenz (Konstante 5)(Variable "x"))
⇝ "Differenz␣(Konstante␣5)␣(Variable␣\"x\")"
```

Beachten Sie, dass Anführungsstriche (") in Zeichenkettenliteralen als (\") dargestellt werden.

9. eigene Klasse

Wie betrachten folgende Klassendeklaration:

```
class Spiegelbar wert where
  gespiegelt :: wert -> wert
```

Instanzen von `Spiegelbar` sollen diejenigen Strukturen sein, die man spiegeln kann. Die Funktion `gespiegelt` liefert zu einem Wert die entsprechende gespiegelte Variante.

i).

Ein Beispiel für spiegelbare Strukturen sind Listen. Die gespiegelte Liste zu einer Liste l enthält die Elemente von l in umgekehrter Reihenfolge. Instanziieren Sie `Spiegelbar` entsprechend.

ii).

Der Typ blattmarkierter Binärbäume lässt sich folgendermaßen definieren:

```
data Baum el = Blatt el | Verzweigung (Baum el)(Baum el)
```

Auch blattmarkierte Bäume lassen sich spiegeln, nämlich indem man für jeden inneren Knoten dessen beide Unterbäume vertauscht. Schreiben Sie eine entsprechende instance-Deklaration.

10. Babylonisches Wurzelziehen

Das Babylonische Wurzelziehen ist eine Methode zur Berechnung von Quadratwurzeln. In verallgemeinerter Form kann es auch zum Ziehen anderer Wurzeln benutzt werden. Es seien x und y reelle Zahlen mit $x > 0$ und $y > 0$. Wir definieren nun eine Folge $\langle z_i \rangle$ wie folgt:

- Das erste Folgenglied z_0 ist eine beliebige positive reelle Zahl.
- Für jedes weitere Folgenglied z_{n+1} gilt:

$$z_{n+1} = \frac{(x-1) \cdot z_n^x + y}{x \cdot z_n^{(x-1)}}$$

Die Folge $\langle z_i \rangle$ nähert sich $\sqrt[x]{y}$ an. Man berechnet nun solange Folgenglieder, bis der Abstand der letzten beiden berechneten Glieder kleiner als eine vorgegebene Genauigkeitsschranke ϵ ist, wobei ϵ eine positive reelle Zahl ist. Schreiben Sie eine Funktion `wurzel`, die die Zahlen ϵ, x, y und z_0 in dieser Reihenfolge als Argumente erwartet und daraus die entsprechende Wurzelnäherung berechnet.

11. abstrakter Datentyp für Mengen

Implementieren Sie einen abstrakten Datentyp `Menge`. Verwenden Sie intern die Darstellung von Mengen als aufsteigend sortierte Listen aus Aufgabe 1. Der abstrakte Datentyp soll folgende Operationen bereit stellen:

Bezeichner	Typ	Bedeutung
leer	`Menge a`	die leere Menge
einfuegen	`a -> Menge a -> Menge a`	Einfügen eines Elements in eine Menge
loeschen	`a -> Menge a -> Menge a`	Löschen einesWerts aus einer Menge
vereinigung	`Menge a -> Menge a -> Menge a`	Vereinigung zweier Mengen
schnitt	`Menge a -> Menge a -> Menge a`	Schnitt zweier Mengen
differenz	`Menge a -> Menge a -> Menge a`	Differenz zweier Mengen
istLeer	`Menge a -> Bool`	Test, ob eine Menge leer ist
istElement	`a -> Menge a -> Bool`	Test, ob einWert Element einer Menge ist
istTeilmenge	`Menge a -> Menge a -> Bool`	Test, ob eine Menge Teilmenge einer zweiten Menge ist
istEchteTeilmenge	`Menge a -> Menge a -> Bool`	Test, ob eine Menge echte Teilmenge einer zweiten Menge ist
minimalesElement	`Menge a -> Maybe a` `Menge a -> a`	das minimale Element einer Menge
maximalesElement	`Menge a -> Maybe a` `Menge a -> a`	das maximale Element einer Menge

Die Funktion `loeschen` soll die ursprüngliche Menge zurückgegeben, wenn der angegebene Wert nicht Element dieser Menge ist. Die Funktionen `minimalesElement` und `maximalesElement` sollen mit einer Fehlermeldung abbrechen oder den Typ `Maybe` verwenden und `Nothing` zurückgeben, wenn die gegebene Menge leer ist.

i).

Legen Sie ein Modul `Menge` für den abstrakten Datentyp an. Fügen Sie in dieses Modul die Deklaration des Datentyps `Menge` sowie die Typsignaturen der Mengenoperationen ein. Legen Sie eine geeignete Exportliste an. Es soll außerdem möglich sein, Mengen auf Gleichheit zu testen und in der üblichen Notation $\{x_1, \ldots, x_n\}$ auszugeben. Sorgen Sie für entsprechende Typklassen Instanziierungen.

ii).

Implementieren Sie die Mengenoperationen. Sie können dazu Ihre Lösung von Aufgabe 1 verwenden.

12. Funktionen höherer Ordnung

i).

Programmieren Sie eine Haskell-Funktion

```
summe :: (Num zahl)=>[zahl] -> zahl
```

welche die Summe der Elemente einer Liste von Zahlen berechnet. Verwenden Sie dazu die vordefinierte Funktion `foldr`.

```
foldr :: (a -> b -> b)-> b -> [a] -> b
```

a und b können gleiche Typen sein.
alles Weitere finden Sie unter: www.haskell.org/hoogle

ii).

Programmieren Sie eine Haskell-Funktion

```
produkte :: (Num zahl)=>[zahl] -> [zahl] -> [zahl]
```

welche zu zwei Listen $[x_1, x_2, \ldots, x_n]$ und $[y_1, y_2, \ldots, y_n]$ die Liste der Produkte $[x_1 \cdot y_1, x_2 \cdot y_2, \ldots, x_n \cdot y_n]$ berechnet. Geben Sie zwei Lösungen an. Die erste Lösung soll die vordefinierten Funktionen `zip`, `map` und `uncurry` verwenden, die zweite Lösung die vordefinierte Funktion `zipWith`.

```
zip :: [a] -> [b] -> [(a,b)]
map :: (a -> b) -> [a] -> [b]
uncurry :: (a -> b -> c) -> (a, b) -> c

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

iii).

Wir stellen einen Vektor als Liste seiner kartesischen Koordinaten dar:

```
type Vektor koord = [koord]
```

Programmieren Sie eine Haskell-Funktion

```
skalarProdukt :: (Num koord)=>Vektor koord -> Vektor koord -> koord
```

welche zu zwei Vektoren $(x_1, \ldots, x_n)$ und $(y_1, \ldots, y_n)$ das Skalarprodukt $\sum_{i=1}^{n} x_i \cdot y_i$ berechnet. Verwenden Sie die in den Teilaufgaben 1 und 2 definierten Funktionen.

iv).

Wir stellen eine Matrix als Liste ihrer Zeilen dar:

```
type Matrix koord = [Vektor koord]
```

Programmieren Sie eine Haskell-Funktion

```
lineareAbb :: (Num koord)=>Matrix koord -> (Vektor koord -> Vektor koord)
```

die zu einer gegebenen $(n \times m)$-Matrix die durch sie dargestellte lineare Abbildung ermittelt, welche m-dimensionale in n-dimensionale Vektoren überführt.

13. Falten von Listen

i).

Im Prelude sind die folgenden Funktionen definiert:

```
(++)   :: [el] -> [el] -> [el]
concat :: [[el]] -> [el]
length :: [el] -> Int
filter :: (el -> Bool) -> [el] -> [el]
```

Geben Sie zu jeder dieser Funktionen einen äquivalenten Haskell-Ausdruck an, der keine **let**-Ausdrücke enthält und höchstens die folgenden vordefinierten Funktionen benutzt:

```
succ  :: Int -> Int
(:)   :: el -> [el] -> [el]
foldr :: (el -> accu -> accu) -> accu -> [el] -> accu
```

ii).

Geben Sie einen zu der Funktion `concat` äquivalenten Haskell-Ausdruck an, der keine **let**-Ausdrücke enthält und höchstens die folgenden vordefinierten Funktionen benutzt:

```
(:)   :: el -> [el] -> [el]
foldl :: (accu -> el -> accu) -> accu -> [el] -> accu
```

14. Varianten von `map` und `foldr` für Bäume

Die Listenfunktionen `map` und `foldr` sind im `Prelude` vordefiniert. Varianten dieser Funktionen gibt es auch für viele andere algebraische Datentypen, so z.B. auch für den Typ `Baum`. Dieser wird wie folgt definiert:

```
data Baum el = Blatt el | Verzweigung (Baum el)(Baum el)
```

Die Typsignaturen der `map`- und der `foldr`-Variante für `Baum` lauten folgendermaßen:

```
baumMap :: (el -> el')-> Baum el -> Baum el'
baumFold :: (el -> accu)-> (accu -> accu -> accu)-> Baum el -> accu
```

Einen Baum `baumMap f baum` erhält man, indem man in `baum` jede Blattmarkierung `el` durch `f el` ersetzt. Einen Baum `baumFold accuBlatt accuVerzweigung baum` erhält man, indem man in `baum` jedes Vorkommen des Datenkonstruktors `Blatt` durch `accuBlatt` und jedes Vorkommen des Datenkonstruktors `Verzweigung` durch `accuVerzweigung` ersetzt. Zum Beispiel liefert

```
baumFold (\el -> [el])(++)((Blatt 2 `Verzweigung` Blatt 3)`Verzweigung` Blatt 5)
```

die Liste `([2] ++ [3])++ [5]`, also `[2, 3, 5]`. Allgemein stellt `baumFold (\el -> [el])(++) baum` die Liste aller Blattmarkierungen von `baum` dar.

i).

Implementieren Sie die beschriebenen Haskell-Funktionen `baumMap` und `baumFold`.

ii).

Programmieren Sie eine Haskell-Funktion

```
hoehe :: Baum el -> Int
```

welche die Höhe eines Baums bestimmt. Die Höhe eines Baums, der nur aus einem Blatt besteht, sei dabei 0, nicht 1. Verwenden Sie für Ihre Implementierung keine **explizite Rekursion**, sondern die Funktion `baumFold`.

15. Sortieren mit variabler Ordnung

Es soll eine Haskell-Funktion `mergeSort` programmiert werden, welche nach verschiedenen Kriterien sortieren kann. Dazu soll ihr neben der zu sortierenden Liste eine Vergleichsfunktion als Argument übergeben werden. Die Vergleichsfunktion soll eine irreflexive Ordnung darstellen.

i).

Welchen Typ hat eine `Vergleichsfunktion` für einen Typ `el`?
Welchen Typ hat dann die Funktion `mergeSort`, die als zusätzliches Argument die zu verwendende Vergleichsfunktion erhält?

ii).

Implementieren Sie die Funktion `mergeSort`, welche eine Vergleichsfunktion sowie die zu sortierende Liste als Argumente bekommt.
als Tipp die „divide and conquer“ Implementierung.

```
dc :: ([a] -> Bool) -> ([a] -> [b]) -> ([a] -> ([a],[a])) ->
  (([b],[b]) -> [b]) -> [a] -> [b]
dc simple solve split combine q
      | simple q  = solve q
      | otherwise   =   combine (l1, l2)
        where
          (p1, p2) = split q
          l1 = dc simple solve split combine p1
          l2 = dc simple solve split combine p2
```

iii).

Betrachten Sie den folgenden Datentyp zur Repräsentation von Büchern:

```
data Buch = Buch {
    autor :: String,
    titel :: String,
    jahr :: Integer,
    verlag :: String
  } deriving (Show)
```

Programmieren Sie die Vergleichsfunktionen `autorVergleich`, `titelVergleich` und `jahrVergleich`, mit denen eine Liste von Büchern nach Autor, Titel bzw. Erscheinungsjahr sortiert werden kann.

iv).

Programmieren Sie eine Haskell-Funktion `und` zur Komposition von Vergleichsfunktionen. Es seien $vergleich_1$ und $vergleich_2$ zwei Vergleichsfunktionen. Dann soll `vergleich1 ‘und‘ vergleich2` eine Vergleichsfunktion sein, mittels der zuerst nach dem durch $vergleich_1$ dargestellten Kriterium und bei äquivalenten Einträgen nach dem durch $vergleich_2$ dargestellten Kriterium sortiert wird. So soll z.B.

```
mergeSort (autorVergleich ‘und‘ titelVergleich)[Liste]
```

Bücher zunächst nach Autor und bei gleichem Autor nach Titel sortieren.

16. Binäre Suchbäume

Wir betrachten den folgenden Datentyp für binäre Suchbäume:

```
data Suchbaum el = Blatt | Knoten (Suchbaum el)el (Suchbaum el)
```

Natürlich koönnen wir mit diesem Datentyp beliebige binäre Bäume darstellen, deren innere Knoten markiert sind. Wir wollen aber nur solche Bäume aufbauen, die auch tatsächlich Suchbäume sind. Dabei gilt folgendes:

- Der leere Baum `Blatt` ist ein Suchbaum
- Ein Baum "`Knoten links wurzel rechts`ïst nur dann ein Suchbaum, wenn folgendes gilt:
  - Die Unterbäume `links` und `rechts` sind Suchbäume.
  - Alle Markierungen von `links` sind kleiner und alle Markierungen von `rechts` sind größer als `wurzel`.

i).

Programmieren Sie eine Haskell-Funktion

```
istElement :: (Ord el)=>el -> Suchbaum el -> Bool,
```

welche testet, ob ein angegebener Wert in einem Suchbaum vorkommt. Nutzen Sie dabei die oben genannten Eigenschaften von Suchbäumen aus, um den Zeitaufwand möglichst gering zu halten.

ii).

Ein Faltungsoperator für den Typ Suchbaum kann wie folgt definiert werden:

```
suchbaumFold :: accu -> (accu -> el -> accu -> accu) -> Suchbaum el -> accu
suchbaumFold accuBlatt accuKnoten = fold where
    fold Blatt = accuBlatt
    fold (Knoten links wurzel rechts) = accuKnoten ( fold links)
                                        wurzel ( fold rechts)
```

Wandeln Sie Ihre Implementierung von `istElement` aus der vorigen Teilaufgabe in eine Implementierung um, welche statt einer expliziten Rekursion `suchbaumFold` verwendet.

17. AVL-Bäume

Wir betrachten den folgenden Datentyp für AVL-Bäume:

```
data AVLBaum el = AVLBlatt | AVLKnoten Int (AVLBaum el)el (AVLBaum el)
```

Verglichen mit den Suchbäumen aus Aufgabe 12 enthalten Knoten eines AVL-Baums zusätzlich eine ganze Zahl, die die Höhe des Baumes angibt. Wir speichern die Höhe statt der Balance, da sich die Höhe bei Änderungen des Baumes einfacher aktualisieren lässt. Die Balance lässt sich aus den Höhen der Unterbäume berechnen.

Vorgaben:

```
hoehe :: AVLBaum el -> Int
hoehe AVLBlatt            = 0
hoehe (AVLKnoten h _ _ _) = h

balance :: AVLBaum el -> AVLBaum el -> Int
balance left right = (hoehe left) - (hoehe right)

avlKnoten :: AVLBaum el -> el -> AVLBaum el -> AVLBaum el
avlKnoten left root right = AVLKnoten h left root right
  where
    h = 1 + max (hoehe left) (hoehe right)

verbinden :: AVLBaum el -> el -> AVLBaum el -> AVLBaum el
verbinden t1@(AVLKnoten h1 l1 w1 r1) r t2@(AVLKnoten h2 l2 w2 r2)
  | balance t1 t2 == 2  && (balance l1 r1) < 0  =
    leftRightRotation (avlKnoten t1 r t2)
  | balance t1 t2 == -2 && (balance l2 r2) > 0  =
    rightLeftRotation (avlKnoten t1 r t2)
  | balance t1 t2 == 2  && (balance l1 r1) >= 0  =
    leftRotation (avlKnoten t1 r t2)
  | balance t1 t2 == -2 && (balance l2 r2) <= 0  =
    rightRotation (avlKnoten t1 r t2)
  | otherwise = avlKnoten t1 r t2
```

```
hoehe :: AVLBaum el -> Int
```

Gibt die Höhe des AVL-Baums zurück.

```
balance :: AVLBaum el -> AVLBaum el -> Int
```

Berechnet die Balance zwischen zwei AVL-Bäumen.

```
avlKnoten :: AVLBaum el -> el -> AVLBaum el -> AVLBaum el
```

Konstruiert aus einem linken Unterbaum, einem Wert und einem rechten Unterbaum einen AVL-Baum, dieser muss nicht balanciert sein.

```
verbinden :: AVLBaum el -> el -> AVLBaum el -> AVLBaum el
```

Vereint zwei AVL-Bäume und ein Element zu einen neuen AVL-Baum. Dabei wird vorausgesetzt, dass die Elemente des ersten AVL-Baums kleiner und die des zweiten AVL-Baums größer als das direkt übergebene Element sind. Normalerweise sollen die beiden AVL-Bäume als linker und rechter Unterbaum und das Element als Wurzel des neuen Baums verwendet werden. Wenn dadurch allerdings ein unbalancierter Baum entstehen würde, soll mittels Rotation bzw. Doppelrotation ein entsprechender balancierter Baum erzeugt werden.

i).

Programmieren Sie die Haskell-Funktionen

```
leftRotation :: AVLBaum el -> AVLBaum el

rightRotation :: AVLBaum el -> AVLBaum el

leftRightRotation :: AVLBaum el -> AVLBaum el

rightLeftRotation :: AVLBaum el -> AVLBaum el
```

welche einen geänderten AVL-Baum durch Rotation balancieren. Die Funktionen `leftRightRotation` und `rightLeftRotation` lasen sich aus den ersten beiden ableiten.

ii).

Programmieren Sie eine Haskell-Funktion

```
avlEinfuegen :: (Ord el)=>el -> AVLBaum el -> AVLBaum el,
```

welche ein Element in einen AVL-Baum einfügt. Wenn der angegebene Wert bereits im AVL-Baum enthalten ist, soll der Baum nicht geändert werden.

iii).

Programmieren Sie eine Haskell-Funktion

```
avlLoeschen :: (Ord el)=>el -> AVLBaum el -> AVLBaum el,
```

welche ein Element aus einem AVL-Baum löscht. Wenn der angegebene Wert nicht im AVL-Baum enthalten ist, soll der Baum nicht geändert werden.