Dieses Dokument ist vom 25.11.2025. Die aktuelle Version des Dokuments kannst du im moodle oder direkt von GitHub herunterladen.

Dieses Dokument enthält 119 Fragen, 13 kleinere bis größere Aufgaben und andere Ressourcen zum Thema Deklarative Programmierung. Die Inhalte dieses Dokuments sollen dir helfen, dein Verständnis über Haskell und Prolog zu prüfen und zu stärken.

Größere Aufgaben haben wir als Challenges markiert. Diese Aufgaben benötigen öfter mehrere Konzepte und führen zusätzlich Konzepte ein, die nur für das Lösen der Aufgabe wichtig sind.

Wenn du Anmerkungen oder weitere Ideen für Inhalte für dieses Dokument hast, dann schreibe uns gerne über z.B. mattermost an – oder erstellt ein issue oder stellt eine PR auf GitHub.

Funktionale Programmierung

Test 1 Was bedeutet es, wenn eine Funktion keine Seiteneffekte hat?

Test 2 Haskell ist eine streng getypte Programmiersprache. Was bedeutet das?

Test 3 Wenn du eine Schleife in Haskell umsetzen möchtest, auf welches Konzept musst du dann zurückgreifen?

Test 4 In imperativen Programmiersprachen sind Variablen Namen für Speicherzellen, deren Werte zum Beispiel in Schleifen verändert werden können. Als Beispiel betrachte

```
def clz(n):
  k = 0
  while n > 0:
    n //= 2
    k += 1
  return 64 - k

def popcnt(n):
  k = 0
  while n > 0:
    if n % 2 == 1:
      k += 1
    n //= 2
  return k
```

In Haskell sind Variablen keine Namen für Speicherzellen. Wie können wir dieses Programm in Haskell umsetzen? Wo wandert das `k` hin?

Test 5 Auf was müssen wir achten, wenn wir eine rekursive Funktion definieren? Die Antwort ist abhängig von dem, was die Funktion berechnen soll. Denke über die verschiedenen Möglichkeiten nach.

Test 6 Gegeben sei das folgende Haskell-Programm.

```
even :: Int -> Bool
even 0 = True
even n = odd (n - 1)

odd :: Int -> Bool
odd 0 = False
odd n = even (n - 1)
```

- Berechne das Ergebnis von `odd (1 + 1)` händisch.
- Wie sieht der Auswertungsgraph für den Ausdruck `odd (1 + 1)` aus?

- Welcher Pfad entspricht deiner händischen Auswertung?
- Welcher Pfad entspricht der Auswertung wie sie in Haskell stattfindet?
- Welcher Pfad entspricht der Auswertung wie sie in Python sinngemäß stattfindet?

Test 7 Es wird als sauberer Programmierstil angesehen, Hilfsfunktionen, die nur für eine Funktion relevant sind, nicht auf der höchsten Ebene zu definieren. Mithilfe welcher Konstrukte kannst du diese lokal definieren?

Test 8 Das Potenzieren einer Zahl x (oder eines Elements einer Halbgruppe) mit einem natürlich-zahligen Exponent n ist in $\mathcal{O}(\log n)$ Laufzeit möglich[1]. Dafür betrachten wir

$$x^n = \begin{cases} \left(x^{\frac{n}{2}}\right)^2 & \text{falls } n \text{ gerade} \\ x \cdot x\left(x^{\frac{n-1}{2}}\right)^2 & \text{sonst} \end{cases}$$

Implementiere eine Funktion, die diese Variante des Potenzierens umsetzt.

Test 9 Gegeben ist folgender Ausdruck.

```
let v = 3
    w = 5
    x = 4
    y = v + x
    z = x + y
 in y
```

Welche Belegungen der Variablen werden tatsächlich berechnet, wenn wir `y` ausrechnen?

Test 10 Ist der folgende Ausdruck typkorrekt?

```
if 0 then 3.141 else 3141
```

Test 11 Wie werden algebraische Datentypen in Haskell definiert?

Test 12 Was ist charakterisierend für Aufzählungstypen, einen Verbundstypen und einem rekursiven Datentypen? Gebe Beispiele für jeden dieser Typarten an.

Test 13 Geben ist der Typ `IntList` mit `data IntList = Nil | Cons Int IntList`. Weiter kann mithilfe der Funktion

```
lengthIntList :: IntList -> Int
lengthIntList Nil         = 0
lengthIntList (Cons _ xs) = 1 + lengthIntList xs
```

die Länge einer solchen Liste berechnet werden. Du möchtest nun auch die Längen von Listen berechnen, die Buchstaben, Booleans oder Gleitkommazahlen enthalten. Was stört dich am bisherigen Vorgehen? Kennst du ein Konzept mit dessen Hilfe du besser an dein Ziel kommst?

Test 14 Wie ist die Funktion `lengthIntList :: IntList -> Int` aus dem vorherigen Test definiert?

Test 15 Du hast einen Datentypen definiert und möchtest dir Werte des Typen nun z.B. im GHCi anzeigen lassen. Was kannst du tun, um an dieses Ziel zu kommen?

Test 16 Wie definieren wir Funktionen?

Test 17 Gebe ein Listendatentypen an, für den es nicht möglich ist, kein Element zu enthalten.

[1] Binäre Exponentiation

Test 18 In Programmiersprachen wie Java greifen wir Daten komplexer Datentypen zu, indem wir auf Attribute von Objekten zugreifen oder getter-Methoden verwenden. Wie greifen wir auf Daten in Haskell zu?

Test 19 Wie sieht eine Datentypdefinition im Allgemeinen aus?

Test 20 Welchen Typ haben

- `(:)` und `[]`,
- `Just` und `Nothing`,
- `Left` und `Right`?

Test 21 Was ist parametrischer Polymorphismus?

Test 22 Welche Typkonstruktoren des kinds `* -> *` kennst du?

Test 23 Welchen kind hat `Either a`?

Test 24 Beim Programmieren vernachlässigen redundante Syntax. Gibt es einen Unterschied zwischen `f 1 2` und `f(1, 2)`

Test 25 Welches Konzept erlaubt es uns, dass wir Funktionen auf Listen nicht für jeden konkreten Typen angeben müssen?

Test 26 Wie gewinnt man aus einem Typkonstruktor einen Typ?

Test 27 Visualisiere `[1, 2, 3]` als Baum, wie du es in der Vorlesung kennengelernt hast. Zur Erinnerung: die inneren Knoten sind Funktionen und die Blätter Werte, die nicht weiter ausgerechnet werden können.

Test 28 Ist `[32, True, "Hello, world!"]` ein valider Haskell-Wert? Warum ja oder nein?

Test 29 Was ist der Unterschied zwischen einem Typ und einem Typkonstruktor?

Test 30 Gegeben ist

```
data Pair a b = Pair a b
```

Wie unterscheidet sich der Typ von

```
data Pair a = Pair a a
```

Challenge 1

- Der größte gemeinsamen Teiler (ggT) zweier Ganzzahlen kann mithilfe des euklidschen Algorithmus berechnet werden. Implementiere das Verfahren.

$$\gcd(x, y) = \begin{cases} |x| & \text{falls } y = 0 \\ \gcd(y, x \bmod y) & \text{sonst} \end{cases}$$

- Alternativ kann der ggT auch berechnet werden, indem wir das Produkt des Schnittes der Primfaktorzerlegung der beiden Zahlen betrachten, also

$$\prod(\mathrm{PF}(x) \cap \mathrm{PF}(y))$$

  wobei PF die Menge der Primfaktoren der gegebenen Zahl (mit entsprechenden Mehrfachvorkommen) beschreiben soll. Implementiere diesen Ansatz.

Challenge 2 Die Ableitung einer Funktion $f : \mathbb{R} \to \mathbb{R}$ kann mithilfe des Differenzenquotienten $\frac{f(x+h)-f(x)}{h}$ für kleines h approximiert werden. Ein andere Methode zur Berechnung der Ableitung ist symbolisches Differenzen und ähnelt dem, wie wir analytisch Ableitungen berechnen. Eine Funktion sei dargestellt durch den folgenden Typ:

```haskell
data Fun = X            -- x     (Variable x)
         | E            -- e     (Euler's constant)
         | Num Double   -- c     (Constant)
         | Ln Fun       -- ln    (Natural logarithm)
         | Fun :+: Fun  -- f + g (Addition)
         | Fun :-: Fun  -- f - g (Subtraction)
         | Fun :*: Fun  -- f * g (Multiplication)
         | Fun :/: Fun  -- f / g (Division)
         | Fun :<: Fun  -- f o g (Composition)
         | Fun :^: Fun  -- f ^ g (Exponentiation)

-- Example
f :: Fun
f = (E :^: X) :<: (X :*: X)  -- (e^x) o (x * x) = e^(x^2)

-- Example
g :: Fun
g :: let x = X
         x2 = x :*: x
         x3 = x2 :*: x
     in x3 :+: x2 :+: x :+: Num 1.0  -- x^3 + x^2 + x + 1
```

- Implementiere eine Funktion `($$) :: Fun -> Double -> Double`, die eine gegebene Funktion in einem gegebenen Punkt auswertet.
- Implementiere eine Funktion `derive :: Fun -> Fun`, die eine gegebene Funktion ableitet.[2]

Challenge 3 In Einführung in die Algorithmik hast du verschiedene Varianten des `mergesort`-Algorithmus kennengelernt. Eine davon hat ausgenutzt, dass in einer Eingabeliste bereits aufsteigend sortierte Teillisten vorkommen können, um den Algorithmus zu beschleunigen.[3] Implementiere diese Variante in Haskell.

Für den Anfang kannst du annehmen, dass die Eingabelisten vom Typ `[Int]` sind. Wenn wir Typklassen behandelt haben, kannst du `Ord a => [a]` nutzen.

Challenge 4 Entwickle einen Datentyp `Ratio`, um rationale Zahlen

$$\frac{p}{q} \in \mathbb{Q}, \quad p \in \mathbb{Z}, q \in \mathbb{N}, p \text{ und } q \text{ teilerfremd}$$

darzustellen. Implementiere die Operationen: Addition, Subtraktion, Multiplikation, Divison. Implementiere weiter eine Funktion, die die rationale Zahl als reelle Zahl mit einer festen Anzahl von Nachkommastellen darstellt.

Test 31 Wie können wir es hinkriegen, dass die invalide Liste `[32, True, "Hello, world!"]` ein valider Haskell-Wert wird? Mithilfe welches Hilfstypen kriegen das hin?

Test 32 Du hast bereits viele Funktionen kennengelernt, die in der Haskell `base`-library implementiert sind. Anstatt eine konkrete Liste dieser Funktionen anzugeben, möchten wir dich motivieren, folgende Dokumentationen verschiedener Module anzuschauen.

- Prelude
- Data.List

Wenn du merkst, die Implementierung einer bekannten Funktion fällt dir ad hoc nicht ein, nehme dir Zeit und überlege, wie du sie implementieren könntest.

[2] Zusammenfassung der Ableitungsregeln

[3] Falls du interessiert bist: In der Haskell `base`-library wird `sort` aus `Data.List` sehr ähnlich implementiert: Data.List.sort.

Test 33 Hier ist eine fehlerhafte Implementierung eines Datentyps für einen knotenbeschrifteten Binärbäumen.

```
data Tree a = Empty | Node Tree a Tree
```

Was ist der Fehler?

Test 34 In imperativen Programmierung iterieren wir über Listen oft in folgender Form (in Java).

```
List<Integer> a = new ArrayList<>();
a.add(3); a.add(1); a.add(4); a.add(1); a.add(5);

List<Integer> b = new ArrayList<>();
for (int i = 0; i < a.size(); i++) {
  b.add(2 * a.get(i));
}
```

Wenn wir diesen Code naiv in Haskell übersetzen, könnten wir z.B.

```
double :: [Int] -> Int -> [Int]
double xs i | i < length xs = 2 * xs !! i : double xs (i + 1)
            | otherwise     = []
```

Das wollen wir niemals so tun.

- Wie unterscheiden sich die Laufzeiten?
- Optimiere die Funktion `double`, sodass sie lineare Laufzeit in der Länge der Liste hat.

Test 35 Die `(!!)`-Funktion ist unsicher in dem Sinne, dass sie für invalide Listenzugriffe einen Fehler wirft. Die Funktion `(!?) :: [a] -> Int -> Maybe a` ist eine sichere Variante von `(!!)`. Sie macht den Fehlerfall explizit durch die Wahl des Ergebnistypen. Was tut diese Funktion voraussichtlich? Implementiere diese Funktion.[4]

Test 36 `(++) :: [a] -> [a] -> [a]` wird verwendet, um zwei Listen aneinanderzuhängen. Wenn wir eine Funktion induktiv über den Listentypen definieren wie z.B. `square :: [Int] -> [Int]`, die jeden Listeneintrag quadrieren soll, dann können wir das wie folgt tun.

```
square :: [Int] -> [Int]
square []     = []
square (x:xs) = [x * x] ++ square xs
```

Die Funktion ist zwar korrekt aber nicht Haskell-idiomatisch, d.h., eine Person, die Erfahren im Programmieren von Haskell ist, würde dies nicht so schreiben. Was müssten wir an der Funktion ändern, damit sie idiomatisch ist.

Test 37 Die Funktion `show` kann genutzt werden, um Werte eines beliebigen Datentyp in eine String-Repräsentation zu überführen. Warum kann `show` nicht als Funktion vom Typ `a -> String` implementiert sein?

Challenge 5 In den Übungsaufgaben hast du einen Suchbaum ohne Höhenbalancierung implementiert. Die Rotationen für einen AVL-Baum lassen sich durch das pattern matching in Haskell vergleichsweise elegant implementieren - erinnere dich z.B. an die Implementierung aus Einführung in die Algorithmik, die recht verbos ist.

Die Höhe eines Teilbaums kann z.B. als weiteres Attribut im Knoten gespeichert werden. Eine ineffizientere Variante ist es, die Höhe mit einer Funktion wiederkehrend zu berechnen. Letztere Variante ist für den Anfang übersichtlicher.

[4] Diese Funktion ist auch bereits vorimplementiert: `(!?)` in `Data.List`.

Implementiere eine Funktion `rotate :: SearchTree a -> SearchTree a`, die einen Teilbaum an der Wurzel rebalanciert, sollte der Teilbaum unbalanciert sein. Diese Funktion kannst du dann nutzen, um die gängigen Operationen auf Suchbäumen anzupassen.[5]

Test 38 Formuliere QuickCheck-Eigenschaften, die die Funktionen

- `isElem :: Int -> SearchTree Int -> Bool`,
- `toList :: SearchTree Int -> Int`,
- `insert :: Int -> SearchTree Int -> SearchTree Int` und
- `delete :: Int -> SearchTree Int -> SearchTree Int`

erfüllen sollen. `isElem` überprüft, ob eine Ganzzahl in gegebenen Suchbaum enthalten ist. `toList` konvertiert einen Suchbaum in eine Liste. `insert` fügt eine Ganzzahl in einen Suchbaum ein. `delete` löscht eine Ganzzahl aus einen Suchbaum.

Wie kannst du die Suchbaum-Eigenschaft spezifizieren (dafür brauchst du weitere Funktionen)?

Test 39 QuickCheck-Eigenschaften werden mit zufällig generierten Werten getestet. Hin und wieder kommt es vor, dass diese Werte Vorbedingungen erfüllen müssen, damit wir Eigenschaften von Funktionen testen können. Wie können wir das erreichen?

Test 40 Wie können wir eine Funktionen teilweise auf Korrektheit testen – also wie können wir für eine beliebige Eingabe verifizieren, dass die Ausgabe korrekt ist?

Test 41 Was sind Funktionen höherer Ordnung?

Test 42 Wie definieren wir Lambda-Abstraktionen bzw. anonyme Funktionen?

Test 43 Warum ist der Typ `(a -> b) -> c` nicht identisch zum Typ `a -> b -> c`? Welcher andere Typ ist identisch zu letzterem?

Test 44 Mit welchen Konzepten gehen die Linksassoziativität der Funktionsapplikation und die Rechtsassoziatvität des Typkonstruktors `(->)` gut Hand in Hand?

Test 45 Zu welchen partiell applizierten Funktionen verhalten sich folgenden Funktionen identisch?

- `succ :: Int -> Int` (die Inkrementfunktion)
- `pred :: Int -> Int` (die Dekrementfunktion)
- `length :: [a] -> Int`
- `sum :: [Int] -> Int`
- `product :: [Int] -> Int`

Test 46 Was ist partielle Applikation?

Test 47 Was ist Currying?

Test 48 Welche Funktionen höherer Ordnung hast du kennengelernt im Kontext der generischen Programmierung? Was ist das Ziel dieser Funktionen?

Test 49 Die Funktionen `map :: (a -> b) -> [a] -> [b]` und `filter :: (a -> Bool) -> [a] -> [a]` lassen sich alle mithilfe `foldr :: (a -> r -> r) -> r -> [a] -> r` ausdrücken. Wie erreichen wir dies?

Test 50 Gegeben seien folgende Funktionen:

- `rgbToHsv :: RGB -> HSV`, die eine Farbe von einer Darstellung in einen anderen konvertiert, und

[5] Rebalancierung eines AVL-Baum

- `hue :: HSV -> Float`, die den Farbwert einer Farbe im Wertebereich $[0°, 360°)$ im HSV-Farbraum zurückgibt.

Du bekommst als Eingabe einen Bild, das hier als Liste von `RGB`-Werten dargestellt ist. Jeder `RGB`-Wert korrespondiert zu einem Pixel. Schreibe eine Funktion, die berechnet, wie viele blaue Pixel das Bild hat. Hier bezeichene eine Farbe als blau, wenn ihr Farbwert zwischen 200° und 250° (inklusiv) liegt. Nutze für die Definition der Funktion sowohl `map` als auch `filter`.

Test 51 Was sind sections im Kontext von Funktionen höherer Ordnung?

Test 52 Welche der Faltungsfunktion auf Listen ergibt sich aus dem Verfahren zur Erzeugung von Faltungsfunktionen, das du für beliebige Datentypen kennengelernt hast?

Test 53 Was ist der Unterschied zwischen `foldl` und `foldr`? Wann liefern `foldl` und `foldr` das gleiche Ergebnis?

Test 54 Wie gewinnen wir aus `foldr` die Identitätsfunktion auf Listen? In den Übungen hast du gelernt, wie man Werte anderer Typen falten kann. Wie gewinnt man aus diesen Funktionen die Identitätsfunktionen auf den jeweiligen Typen?

Test 55 Gegeben sind folgende Datentypen
- `data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)`,
- `data Rose a = Node [Rose a]`.

Welche Typen haben die jeweiligen Datenkonstruktoren und wie führen wir diese in die Signatur der jeweiligen Faltungsfunktion über? Wo benötigen wir rekursive Aufrufe der jeweiligen Faltungsfunktionen?

Test 56 Betrachte die Funktion

```
f :: [a] -> b
f []     = e
f (x:xs) = g x (f xs)
```

Nach diesem induktiven Muster sind viele Funktionen auf Listen implementiert. Nehme als Beispiel die Funktion `sum :: [Int] -> [Int]`.

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

Dieses Muster haben wir in `foldr` abstrahiert. Wo wandern die jeweiligen Bestandteile der abstrakten Funktion `f` hin, wenn wir `f` mithilfe von `foldr` definieren. Was passiert insbesondere mit dem rekursiven Aufruf von `f`?

Test 57 Wie kannst du mithilfe von Faltung viele Elemente in einen Suchbaum einfügen oder lösen? Implementiere
- `insertMany :: [Int] -> SearchTree Int -> SearchTree Int` und
- `deleteMany :: [Int] -> SearchTree Int -> SearchTree Int`.

Du kannst davon ausgehen, dass du die Einfüge- und Löschfunktion für einzelne Elemente bereits hast.

Test 58 Wir können `map :: (a -> b) -> [a] -> [b]` mithilfe von `foldr` wie folgt implementieren:

```
map f xs = foldr (\x ys -> f x : ys) [] xs
```

Vereinfache den Lambda-Ausdruck mithilfe von Funktionen höherer Ordnung.

Test 59 Wenn wir die Listenkonstruktoren in `foldr` einsetzen, erhalten wir die Identitätsfunktion auf Listen, also

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

Wenn wir das Gleiche mit `foldl` und angepassten `(:)` machen, also

```
foldl (flip (:)) [] :: [a] -> [a],
```

dann erhalten wir nicht die Identitätsfunktion auf Listen. Warum und was bekommen wir stattdessen heraus?

Test 60 Es gibt viele andere hilfreiche Funktionen höherer Ordnung in der Haskell Prelude. Eine von diesen ist `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`. Sie verknüpft jeweils zwei Elemente aus den jeweiligen Listen unter der gegeben Funktion.

- Implementiere `zipWith` mithilfe von `map, uncurry, zip`.
- Implementiere `zip` mithilfe von `zipWith`.
- Implementiere das Prädikat `isSorted` mithilfe von `zipWith`.

Challenge 6 Gegeben sei die Funktion Faltungsfunktion `foldTree :: (r -> a -> r -> r) -> r -> Tree a -> r` für einen knotenbeschrifteten Binärbaum gegeben durch `data Tree a = Empty | Node (Tree a) a (Tree a)`.

Wie auch für Listen lassen sich eine Reihe von bekannten Funktionen auf Bäume übertragen.[6] Implementiere die Funktionen

- `any :: (a -> Bool) -> Tree a -> Bool` und `and :: (a -> Bool) -> Tree a -> Bool`,
- `elem :: Int -> Tree Int -> Bool` und `notElem :: Int -> Tree Int -> Bool`,
- `toList :: Tree a -> Bool`,
- `null :: Tree a -> Bool` (überprüft, ob der Baum leer ist),
- `length :: Tree a -> Int`,
- `maximum :: Tree Int -> Int` und `minimum :: Tree Int -> Int`, und
- `sum :: Tree Int -> Int` und `product :: Tree Int -> Int`.

Test 61 Gegeben seien die Funktion

```
f :: a -> b
g :: a -> b -> c
```

sowie die Kompositionsfunktion `(.) :: (b -> c) -> (a -> b) -> a -> c`.

In der Typdefinition von `(.)` scheint das erste Argument, eine einstellige Funktion zu sein. Ist der Ausdruck

```
g . f
```

trotzdem typkorrekt, obwohl `g` eine zweistellige Funktion ist? Wenn ja, wie werden die Typvariablen – insbesondere das `c` der Komposition – unifiziert?

Test 62 Gegeben sei der Datentyp `data Tree a = Empty | Node (Tree a) a (Tree a)` und die Faltungsfunktion `foldTree :: r -> (r -> a -> r -> r) -> Tree a -> r`.

Vergewissere dich, dass die Implementierung der folgenden Funktion, die alle Beschriftungen durch den gleichen Wert ersetzt, korrekt ist.

```
replace :: b -> Tree a -> Tree b
replace x = foldTree Empty (const . flip Node x)
```

[6] Diese Funktionen lassen sich auf alle faltbaren Datentypen verallgemeinern. Für Interessierte: Dies wird mithilfe der Typklasse `Foldable` festgehalten – diese Typklasse behandeln wir aber in der Vorlesung voraussichtlich nicht.

Zeige, dass `const . flip Node x = \l y r -> Node l x r` ist.

Test 63 Welche Funktion verbirgt sich hinter `foldr ((++) . f) []` und was ist ihr Typ?

Test 64 Versuche in den folgenden Ausdrücken, Teilausdrücke schrittweise durch bekannte Funktionen zu ersetzen und gegebenenfalls zu vereinfachen.

- `foldr (\x ys -> f x : ys) [] (foldr (\x ys -> g x : ys) [] xs)`,
- `map (\_ -> y) xs`,
- `foldr (\x ys -> if x `mod` 2 == 1 then x - 1 : ys else ys) [] xs`,
- `foldl (\ys x -> x : ys) [] xs` und
- `flip (curry fst) x`.[7]

Challenge 7 Sei $m, n \in \mathbb{N}$. Gegeben seien

- die Identitätsfunktion auf $\{0, ..., n-1\}$ mit $\pi_0 : \{0, ..., n-1\} \to \{0, ..., n-1\}, x \mapsto x$ und
- eine endliche Folge von Paaren $((a_i, b_i))_{i \in \{1,...,m\}}$ mit $a_i \in \{0, ..., n-1\}, b_i \in \{0, ..., n-1\}$ für alle $i \in \{1, ..., m\}$.

Wir definieren $\pi_{i,j}$ als

$$\pi_{i,j}(k) = \begin{cases} i & \text{falls } k = j \\ j & \text{falls } k = i \\ k & \text{sonst} \end{cases}$$

für alle $k \in \{0, ..., n-1\}$.

Wir betrachten die Paare als Vertauschungen der Bilder der Abbildung π_0, d.h.,

$$\pi_i = \pi_{a_i, b_i} \circ \pi_{i-1} \text{ für } i \in \{1, ..., m\}.$$

- Implementiere eine Funktion `swaps :: Int -> [(Int, Int)] -> [Int]`, die π_m mithilfe von Listen berechnet. Der erste Parameter bestimmt die Menge $\{0, ..., n-1\}$ und der zweite die Folge.
- Implementiere eine Funktion `swaps :: Int -> [(Int, Int)] -> Int -> Int`, die π_m mithilfe von Funktion berechnet. (Hier ist der erste Parameter unter Umständen redundant.)
- Welche Vor- und Nachteile haben die jeweiligen Ansätze im Vergleich?

Test 65 Eta-reduziere die folgende Ausdrücke:

- `sum xs = foldr (+) 0 xs`,
- `add a b = a + b` und
- `\x ys -> (:) x ys`.

Test 66 Implementiere die Funktion `insert :: Int -> a -> Map Int a -> Map Int a`, die ein Schlüssel-Wert-Paar in eine `Map Int a` einfügt. Die `Map` ist wie folgt repräsentiert `type Map k v = k -> v`.

Test 67 Wir haben `foldr :: (a -> b -> b) -> b -> [a] -> b` als natürliche Faltungsfunktion kennengelernt, die einen Ausdruck erzeugt, der rechts geklammert ist. Zum Beispiel gilt

`foldr (+) 0 [1, 2, 3] = 1 + (2 + (3 + 0))`.

Das gleiche Ziel können wir mit anderen Typen verfolgen. Implementiere eine Funktion `foldr :: (a -> b -> b) -> b -> Tree a -> b` für einen blattbeschrifteten Binärbaum `data`

[7] „Your scientists were so preoccupied with whether or not they could, that they didn't stop to think if they should." Jenseits solcher kleinen Verständnisfragen gilt weiterhin, dass wir verständlichen Code schreiben wollen. Solche Ausdrücke sind häufig schwieriger zu verstehen – auch wenn es unterhaltsam ist, sich solche Ausdrücke auszudenken.

`Tree a = Leaf a | Tree a :+: Tree a`, die den gleichen Ausdruck erzeugt. Zum Beispiel soll

```
foldr (+) 0 ((Leaf 1 :+: Leaf 2) :+: Leaf 3) = 1 + (2 + (3 + 0))
```

gelten.

Test 68 Gegeben sei folgendes Python-Programm.

```
from dataclasses import dataclass
from typing import Generic, TypeVar


class Foldable():
  def foldr(self, f):
    pass

  def sum(self):
    return self.foldr(lambda x: lambda ys: x + ys)(0)

  def toList(self):
    return self.foldr(lambda x: lambda ys: [x] + ys)([])

  def __len__(self):
    return self.foldr(lambda _: lambda s: 1 + s)(0)

  def __contains__(self, y):
    return self.foldr(lambda x: lambda z: z or x == y)(False)

  # ...


T = TypeVar('T')

class Tree(Generic[T], Foldable):
  def foldr(self, f):
    def foldr_with_f(e):
      match self:
        case Empty():
          return e
        case Node(l, x, r):
          y = f(x)(r.foldr(f)(e))
          z = l.foldr(f)(y)
          return y
    return foldr_with_f

@dataclass
class Empty(Tree[T]):
  pass

@dataclass
class Node(Tree[T]):
  left: Tree[T]
  value: T
  right: Tree[T]


tree = Node(Empty(), 3, Node(Node(Empty(), 7, Empty()), 4, Empty()))
```

```
print(tree.sum())  # 14
print(tree.toList())  # [3, 7, 4]
print(len(tree))  # 3
print(3 in tree, 9 in tree)  # True False
```

In diesem Programm werden viele Konzepte verwendet, die du im Haskell-Kontext kennengelernt hast – aber wahrscheinlich bisher nicht in Python gesehen hast. In diesem Test geht es darum, die diese Konzepte zu identifizieren.

Wo findest du
- Funktionen höherer Ordnung,
- pattern matching,
- algebraische Datentypen (Typkonstruktoren, Datenkonstruktoren),
- parametrischen Polymorphismus[8],
- ad-hoc Polymorphismus (Typklassen bzw. Überladung) und
- lokale Definitionen.

Data classes und match statements brauchst du dir jenseits dieses Tests nicht anschauen (wenn es dich nicht weiter interessiert). Es soll in dem Test nur darum gehen, die Haskell-Konzepte zu erkennen. In Bemerkung 1 kannst du das gleiche Programm in Java sehen.

Test 69 In das folgende Python-Programm hat sich ein bug hineingeschlichen.

```
text = 'Ja, ja, ich back mir \'nen Kakao!'
say_words = []

for word in text.split():
  say_words.append(lambda sep: print(word, end=sep))

for say_word in say_words[:-1]:
  say_word(' ')
say_words[-1]('\n')
```

Dieses Programm gibt sieben Mal „Kakao!“ aus. Erkläre wie dieses Verhalten zustande kommt? Wie kannst du den bug beheben? Kann der gleiche Fehler in Haskell passieren?

Test 70 Was ist ein abstrakter Datentyp? Was sind die Bestandteile eines abstrakten Datentyps?

Test 71 Wie definieren wir die Semantik der zu einem abstrakten Datentyp gehörenden Operationen? Wie definieren wir sie insbesondere nicht?

Test 72 Wieso ist das sofortige Nutzen einer Gleichheit auf einem abstrakten Datentypen problematisch? Was sollte man stattdessen tun?

Test 73 Zur Spezifikation der Semantik nutzen wir Gesetze, die bestimmen, wie verschiedene Operationen miteinander interagieren. Dafür benötigen wir verschiedene Werte oftmals unterschiedlicher Datentypen. Wo kommen diese her und wie sind sie quantifiziert?

Test 74 Welche Eigenschaften sollten die für einen abstrakten Datentypen formulierten Gesetze erfüllen, damit sie eine sinnvolle Semantik beschreiben?

Challenge 8 Gebe folgende abstrakte Datentypen an: Paar, Menge, stack, queue, double-ended queue, knotenbeschrifteter Binärbaum, priority queue.

[8] Typannotationen in Python sind nicht sonderlich elegant. Deshalb sind nur die angegeben, um den parametrischen Polymorphismus zu identifizieren und data classes anständig zu nutzen.

Anschließend kannst du diese auch (naiv) implementieren und deine Implementierung testen, indem du deine formulierten Gesetze mit QuickCheck implementierst.

Test 75 Was sind Typklassen?

Test 76 Wie unterscheidet sich der Polymorphismus, der durch Typklassen ermöglicht wird, vom parametrischen Polymorphismus?

Test 77 In einem vorherigen Test wurdest du bereits gefragt, wieso `show` nicht als Funktion mit dem Typ `a -> String` implementiert sein. Wieso wird die Funktion durch den Typ `Show a => a -> String` gerettet?

Test 78 Welche Typklassen kennst du? Was ermöglichen sie konkret?

Test 79 Eine `Show`-Instanz für den Typ `Tree a = Leaf a | Tree a :+: Tree a` könnte wie folgt aussehen:

```
instance Show a => Show (Tree a) where
  show (Leaf x)  = "Leaf " ++ show x
  show (l :+: r) = "(" ++ show l ++ ") :+: (" ++ show r ++ ")"
```

Welche haben Werte von `Tree a` führen zur worst-case Laufzeit und welche zur best-case Laufzeit? Die Anzahl der Blätter soll hier frei sein. Welche Eigenschaften von `(++)` führen zu den jeweiligen Laufzeiten?

Test 80 Überlade die Operationen `(+)`, `(-)`, `(*)`, `abs`, `signum`, `fromInteger` für den Datentypen `data Mat22 a = Mat22 a a a a`, der (2×2)-Matrizen repräsentieren soll – `abs`, `signum`, `fromInteger` kannst du z.B. komponentenweise implementieren.[9]

Test 81 Mit

$$\begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}^n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

und der binären Exponentiation und `Mat22 Integer` aus einem vorherigen Tests kannst du die n-te Fibonacci-Zahl in logarithmischer Laufzeit in n berechnen. Implementiere das Verfahren.

Da du eine `Num`-Instanz auf `Mat22` definiert hast, kannst du den `(^)`-Operator zum binären Exponentiation nutzen.

An vielen Stellen in den bisherigen Selbsttests haben wir oft einen konkreten Typ (z.B. `Int`) genutzt, für den es bestimmte Typklasseninstanzen gibt. Das ist meistens der Fall gewesen, wenn wir Gleichheit auf Werten oder eine Vergleichsoperation auf Werten brauchten. Schau dir die bisherigen Selbsttests erneut an und überlege dir, wo du Typen verallgemeinern kannst.

Test 82 Welche Funktionen musst du implementieren, damit eine `Eq`-Instanz vollständig definiert ist? Welche Gesetze sollten die Funktionen einer `Eq`-Instanz erfüllen?

Test 83 Gegeben sei der Typ

```
data Tree a b c = Empty | Leaf a | Node (Tree a b c) Int c (Tree a b c).
```

Implementiere eine `Eq`-Instanz für diesen Typen. Die Gleichheit soll sich so verhalten, wie die die wir durch das Ableiten bekommen würden. Bevor du die Instanz implementierst, überlege dir:

- Wie viele Regeln brauchst du mindestens, um `(==)` zu definieren?
- Benötigst du für die Implementierung Typeinschränkungen? Wenn ja, für welche Typen?

[9] Oft sind an Funktionen von Typklassen Bedingungen bzw. Gesetze, die erfüllt werden sollen, gekoppelt. Das für `abs` und `signum` wird durch den Vorschlag nicht erfüllt.

- An welchen Stellen wirst du `(==)` rekursiv anwenden?
- Die Datenkonstruktoren sind auf den rechten Seiten der Regeln nicht relevant. Auf welchen Typen kannst du die Gleichheit für z.B. `Node` zurückführen, bzw. wenn du dir die rechte Seite der Regel für `Node` anschaust, welche Typen fallen dir ein, für die diese rechte Seite auch eine Gleichheit definieren würde?

Test 84 Welche Funktionen musst du implementieren, damit eine `Ord`-Instanz vollständig definiert ist? Welche Gesetze sollten die Funktionen einer `Ord`-Instanz erfüllen?

Test 85 Gegeben sei der Typ

```
data Tree a b c = Empty | Leaf a | Node (Tree a b c) Int c (Tree a b c).
```

Implementiere eine `Ord`-Instanz für diesen Typen. Die Ordnung soll sich so verhalten, wie die die wir durch das Ableiten bekommen würden. Bevor du die Instanz implementierst, überlege dir:

- Spielt die Reihenfolge, in der wir die Datenkonstruktoren definieren eine Rolle für die Ordnung? Wenn ja, wie?
- Wie viele Regeln brauchst du mindestens, um `compare` zu definieren? Wie auch bei der Typklasse `Eq` können wir eine Regel definieren, die alle Fälle abdeckt, in denen wir `GT` erhalten, wenn wir uns nur die Datenkonstruktoren anschauen. Welches Schema müssen wir für die anderen Regeln verwenden, damit das funktioniert?
- Benötigst du für die Implementierung Typeinschränkungen? Wenn ja, für welche Typen?
- An welchen Stellen wirst du `compare` rekursiv anwenden?
- Die Datenkonstruktoren sind auf den rechten Seiten der Regeln nicht relevant. Auf welchen Typen kannst du die Ordnung für z.B. `Node` zurückführen, bzw. wenn du dir die rechte Seite der Regel für `Node` anschaust, welche Typen fallen dir ein, für die diese rechte Seite auch eine Ordnung definieren würde?

Implementiere die Ordnung auch erneut mit `(<=)`.

Test 86 Die Typklasse `Ord` ist wie folgt definiert:

```
class Eq a => Ord a where
  compare              :: a -> a -> Ordering
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min             :: a -> a -> a

  -- default definitions
  -- ...

  {-# MINIMAL compare | (<=) #-}
```

Das `{-# MINIMAL compare | (<=) #-}` bedeutet, dass es genügt, entweder `compare` oder `(<=)` zu implementieren.

- Gebe Standarddefinitionen für die Funktionen der Typklasse an.
- Was ermöglicht es, eine Standarddefinition für `compare` angeben zu können?
- Deine Standarddefinition von `compare` ist voraussichtlich ineffizient.[10] Woran liegt das? Mit Hinsicht auf Effizienz – welche der beiden Funktionen würdest du implementieren, wenn du nur eine implementieren dürftest?[11]

[10] Die Vordefinierte ist es auch, also keine Sorge.

[11] Auch wenn Standarddefinitionen für den Anfang hilfreich sind, um mit minimalem Aufwand alle Funktionen einer Typklasse zu verwenden, findet man häufig konkrete Implementierungen für mehr als nur die Funktionen, für die es notwendig ist.

Test 87 In nicht streng getypten Programmiersprachen haben wir oft mit impliziter Typkonversion zu tun.[12] Implementiere eine Funktion `ifThenElse`, die als Bedingung Werte beliebiger Typen entgegennehmen kann. Ziel ist es, dass der folgende Ausdruck ausgewertet werden kann.

```
let a = ifThenElse 0 3 4
    b = ifThenElse [5] 6 7
    c = ifThenElse Nothing 8 9
 in a + b + c  -- 19
```

Test 88 Eine Halbgruppe ist eine Struktur $(H, *)$, wobei $*$ eine assoziative, binäre Verknüpfung $* : H \times H \to H$ ist. Ein Monoid erweitert die Halbgruppe um ein neutrales Element bzgl. $*$.

Definiere Typklassen `Semigroup` und `Monoid`, diese Strukturen implementieren. Gebe auch beispielhaft ein paar Instanzen für diese an.

Test 89 Wo findest du das Konzept der Typklassen in Programmiersprachen wie z.B. Python oder Java wieder? Gibt es z.B. ein Pendant zur `Show`-Typklasse in diesen Programmiersprachen?

Test 90 Was ist Lazy Evaluation?

Test 91 Wie werden Berechnungen in Haskell angestoßen? Wie viel wird berechnet?

Test 92 Gebe ein Beispiel an, das zeigt, dass die faule Auswertung berechnungsstärker ist?

Test 93 Welche praktischen Vorteile ergeben sich aus der Lazy Evaluation?

Test 94 Wie werden mehrfache Berechnungen in einer nicht-strikten Auswertungsstrategie vermieden?

Test 95 Gegeben sei folgendes Haskell-Ausdruck.

```
let c = x == 0
    a = u `div` x
    b = 0
 in if c then b else a
```

Der Teilausdruck `a = u `div` x` erscheint auf dem ersten Blick problematisch, da `x` Null sein könnte. Wieso stellt das mit Lazy Evaluation kein Problem dar?

Test 96 Eine zyklische einfach-verkettete Liste können wir in Python z.B. so definieren.

```
class Node:
  def __init__(self, value):
    self.value = value
    self.next = None

one, two = Node(1), Node(2)
one.next, two.next = two, one
```

Wenn wir mit `one` starten, dann haben wir nun die unendliche Liste `[1, 2, 1, 2, ...]`.

Die Mutierbarkeit des `next`-Zeigers macht das Verlinken der Knoten möglich. Wie können wir in Haskell, trotz der Abwesenheit von Mutierbarkeit, zyklische Datenstrukturen umsetzen? Versuche, dein Programm ähnlich zum Python-Programm aussehen zu lassen.[13]

[12] Diese wollen wir nun für einen Moment nach Haskell zurückholen, um sie dann ganz schnell wieder zu vergessen.
[13] Für Interessierte: Die Technik ist als Tying the Knot bekannt.

Test 97 Gegeben sei der Datentyp

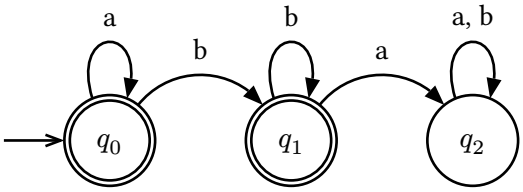
```
data Doubly a = Null | Node (Doubly a) a (Doubly a).
```

- Implementiere eine Funktion `fromList :: [a] -> Doubly a`, die die gegebene Liste in eine doppelt-verkettete Liste umwandelt. `Null` soll sowohl das linke als auch das rechte Ende der Liste darstellen. Von diesem brauchst nicht zum anderseitig verketteten Element zurückkommen. `fromList` soll den Knoten zurückgeben, der mit dem ersten Listenelement korrespondiert.
- Weiter implementiere auch `prev :: Doubly a -> Doubly a`, `value :: Doubly a -> Maybe a` und `next :: Doubly a -> Doubly a`, die den vorherigen Knoten, die Beschriftung eines Knoten, und den nächsten Knoten zurückgeben sollen.
- Angenommen du möchtest einen weiteren Wert in die doppelt-verkettete Liste einfügen, auf welches Problem stoßt du hinsichtlich Mutierbarkeit?

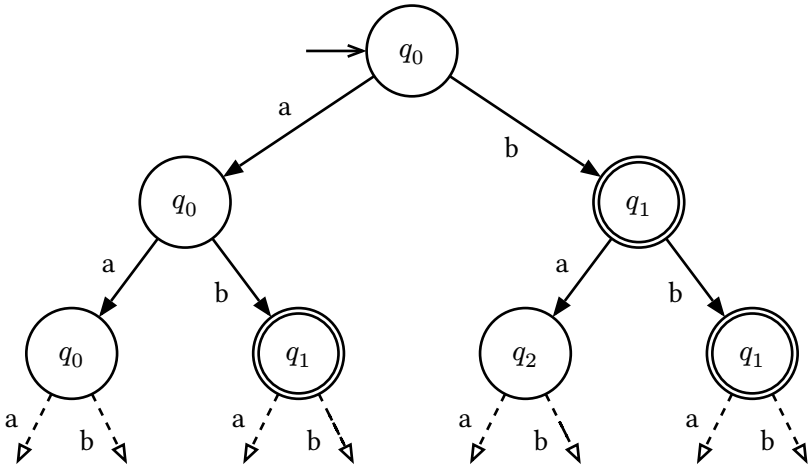
Der Wert des folgenden Ausdrucks soll `8` sein.

```
let xs = fromList [1, 6, 1, 8, 0, 3]
 in value . prev . next . next . next . next $ xs
```

Challenge 9 Wir können endliche Automaten als unendliche Bäume darstellen. Betrachte z.B. den endlichen Automaten für die reguläre Sprache a^*b^*.



Diesen können wir als unendlichen Baum wie folgt darstellen.



- Konstruiere diesen Baum `asbs :: State Char` mithilfe des Typs

  ```
  data State a = State Bool [(a, State a)].
  ```

  Der Boolean gibt an, ob der Zustand akzeptiert, und `[(a, State a)]` gibt die ausgehenden Transitionen an.
- Implementiere eine Funktion `accept :: Eq a => [a] -> State a -> Bool`, die bestimmt, ob eine Eingabe akzeptiert wird.

- Implementiere eine Funktion `language :: State a -> [[a]]`, die die akzeptierte Sprache des Automaten zurückgibt. (Du kannst davon ausgehen, dass die Sprache nicht leer ist – wenn du Entscheidungsproblem trotzdem lösen möchtest, halten wir dich nicht auf.)
- Warum funktioniert die folgende Implementierung der Funktion `language` nicht?

```
language :: State a -> [[a]]
language (State False ts) = [c:ws | (c, q) <- ts, ws <- language q]
language (State True  ts) = [] : [c:ws | (c, q) <- ts, ws <- language q]
```

Test 98 Wieso können wir mit `foldl` auf unendlichen Listen mit keinem Ergebnis rechnen?

Test 99 `scanl :: (b -> a -> b) -> b -> [a] -> [b]` und `scanr :: (a -> b -> b) -> b -> [a] -> [b]` sind ähnlich zu `foldl` und `foldr`. Beide Funktionen speichern die Zwischenergebnisse der jeweiligen Funktion in Listen.

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
scanl _ e []     = [e]
scanl f e (x:xs) = e : scanl f (f e x) xs

-- scanl (+) 0 [1..4] = [0, 1, 3, 6, 10]

scanr :: (a -> b -> b) -> b -> [a] -> [b]
scanr _ e []     = [e]
scanr f e (x:xs) = f x y : ys
  where ys@(y:_) = scanr f e xs

-- scanr (+) (0) [1..4] = [10, 9, 7, 4, 0]
```

- Welche der beiden Funktionen kann auf unendlichen Listen arbeiten?
- (Implementiere die Fibonacci-Folge `fibs :: [Integer]` mithilfe von einer der beiden Funktionen.)

Challenge 10 Fixpunktverfahren sind iterative Methoden, bei denen eine Funktion wiederholt auf einen Wert angewendet wird, bis sich ein stabiler Punkt ergibt, der sich durch weitere Anwendungen nicht mehr verändert.

Dieses Berechnungsmuster wird durch die Funktion `iterate :: (a -> a) -> a -> [a]` in der Prelude festgehalten.

- Implementiere `iterate`.
- Ein klassisches Beispiel aus der Numerik ist die Berechnung der Wurzel mithilfe des Heron-Verfahrens. Es ist gegeben durch

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

  Diese Folge nähert den Wert von $\sqrt{a}$ mit jedem Folgeglied besser an. Implementiere das Verfahren mithilfe von `iterate`. Wähle die erste Nährung x_{n+1}, die $\left|x_{n+1} - x_n\right| < \varepsilon$ erfüllt, für ein gegebenes $\varepsilon > 0$.
- Implementiere die Fibonacci-Folge als unendliche Liste `fibs :: [Integer]` mithilfe von `iterate`. Du brauchst eine Hilfsfunktion, die die Elemente der Ergebnisliste von `iterate` projeziert.
- Solange eine Liste Inversionen enthält, d.h., es existieren i, j mit $i < j$, sodass $a_i > a_j$ gilt, gilt eine Liste als unsortiert. Das schrittweise Entfernen solcher Fehlstellungen führt zu einer sortierten Liste.

- Implementiere eine Funktion `resolve :: Ord a => [a] -> [a]`, die eine Fehlstellung findet und sie auflöst, indem sie die Elemente an den entsprechenden Positionen tauscht.
- Implementiere das daraus resultierende Sortierverfahren mithilfe von `iterate`.

Challenge 11 Die Editierdistanz zwischen zwei Wörtern $u \in \Sigma^m, v \in \Sigma^n$ können wir mithilfe der folgenden Rekurrenz bestimmen:

$$\mathrm{ed}(i,j) = \begin{cases} 0 & \text{falls } (i,j) = (0,0) \\ i & \text{falls } j = 0 \\ j & \text{falls } i = 0 \\ \\ \mathrm{ed}(i-1, j-1) & \text{falls } u_{i-1} = v_{j-1} \\ \min(\mathrm{ed}(i-1,j-1), \mathrm{ed}(i,j-1), \mathrm{ed}(i-1,j)) + 1 & \text{sonst} \end{cases}$$

für alle $0 \leq i \leq m, 0 \leq j \leq n$. Um $\mathrm{ed}(m,n)$ effizient auszurechen, nutzt man dynamische Programmierung. Das heißt, wir merken uns die Zwischenergebnisse und nutzen diese, wenn wir sie erneut brauchen, anstatt sie neu zu berechnen. Dieses Konzept ist auch als memoization bekannt.

In Haskell können wir memoization mithilfe von Lazy Evaluation umsetzen. Hier ist ein unvollständiges Haskell-Programm, dass die Editierdistanz berechnen soll.

```
editdist :: Eq a => [a] -> [a] -> Int
editdist u v = table !! m !! n
  where
    (m, n) = (length u, length v)
    table = [[ed i j | j <- [0..n]] | i <- [0..m]]
```

- Definiere die Funktion `ed :: Int -> Int -> Int` lokal in `editdist`. `ed` soll hier die zwischengespeicherten Ergebnisse aus `table` verwenden.
- Überlege dir wie hier laziness und memoization zusammenspielen.
- Welche worst-case Laufzeit hat dein Problem, wenn du annimst, dass die Laufzeit `(!!)` konstant ist?

Test 100 Gegeben sei der Datentyp

```
data Direction = North | East | South | West.
```

- Implementiere eine `Enum`-Instanz für `Direction`.
- Implementiere eine `Bounded`-Instanz für `Direction`.
- Implementiere die Funktionen `turnLeft :: Direction -> Direction` und `turnRight :: Direction -> Direction`, die die Himmelsrichtungen entsprechend ihrer Bezeichnung durchgehen.
- Implementiere eine Funktion `allDirections :: [Direction]`, die alle Himmelsrichtungen auflistet. Nutze dafür Funktionen, die dir durch die vorherigen Typklassen bereitgestellt werden.

Test 101 Implementiere Funktion `cycleFrom :: (Enum a, Bounded a) => a -> [a]`, die ab einem gegebenen Wert alle Werte des Typen nicht absteigend durchläuft. Wenn größte Wert erreicht ist, soll die Liste wieder beim kleinsten Wert des Typen beginnen.

Test 102 An welches mathematische Konzept sind list comprehensions angelehnt?

Test 103 Aus welchen Teilen besteht eine list comprehension?

Test 104 Implementiere die Funktionen `map`, `filter` und `concatMap` mithilfe von list comprehensions.

Welcome weary traveler, you've come far. Why don't you stop here and get some rest before you continue your journey.

(Mit den Inhalten der Vorlesung kannst du die Tests und Challenges aktuell bis hier hin lösen.)

Test 105 Was ist referenzielle Transparenz?

Test 106 Welche Rolle spielt der Typ `IO a` bzgl. Seiteneffekte? Was beschreibt ein Wert vom Typ `IO a`?

Test 107 Wie können wir zwei `IO`-Aktionen zu einer neuen `IO`-Aktion kombinieren?

Test 108 Betrachte die `IO`-Aktion `act1 >> act2`.
- Welche der beiden Aktionen wird zuerst ausgeführt?
- Warum erscheint das bei Lazy Evaluation kontraintuitiv?
- Welche Rolle spielt der Typ `RealWorld -> (RealWorld, a)` bei der Sequenzierung?

Test 109 Implementiere ein Programm, das Zahlen aus einer Datei aufsummiert, bzw. implementiere eine Funktion `sumFile :: FilePath -> IO Int`. In jeder Zeile einer Datei steht eine nicht-negative Zahl.

Test 110 Implementiere folgendes Rate-Spiel als IO-Programm. Es soll eine Zahl erraten werden.
- Du bekommst ein Orakel vom Typ `a -> Ordering`, das dir verrät, ob dein Rateversuch kleiner als, gleich oder größer als der Wert ist, den das Orakel festgelegt hat.
- In jeder Runde des Spiels ratest du eine Zahl.
- Wenn die Zahl kleiner als die unbekannte Zahl ist, dann soll der Hinweis „Die gesuchte Zahl ist kleiner." ausgegeben werden. Wenn die unbekannte Zahl größer ist, soll ebenso eine entsprechende Nachricht ausgegeben werden.
- Wenn die korrekte Zahl erraten wurde, bricht das Spiel ab.

Implementiere das Spiel als Funktion `game :: Read a => (a -> Ordering) -> IO ()`.

Du kannst das Spiel gerne ausschmücken und erweitern. Zum Beispiel kannst du die Anzahl der Rateversuche begrenzen oder eine weitere Zahl festlegen, die vorzeitig das Spiel beendet und das Orakel gewinnen lässt.

Test 111 Die `Applicative`-Typkonstruktorklasse erlaubt es uns, `fmap` auf Funktionen mit mehreren Argumenten zu verallgemeinern. Dadurch können wir etwa

`(+) <$> Just 1 <*> Just 2` oder `Just (+) <*> Just 1 <*> Just 2`

schreiben. Die Operatoren `(<$>)` und `(<*>)` funktionieren dabei ähnlich wie `($)` – mit `(<$>)` muss die Funktion nicht explizit in den entsprechenden `Applicative` gehoben werden.

Ein Typ, der uns konzeptionell noch näher an die gewöhnliche Funktionsapplikation heranführt, ist

```
newtype Identity a = Identity { runIdentity :: a }.
```

Implementiere `Functor`-, `Applicative`- und `Monad`-Instanzen für `Identity`.

Wenn du die Instanzen definierst, solltest du feststellen, dass du im Wesentlichen nur den enthaltenden Wert aus der `Identity` holst, verarbeitest und anschließend wieder hereinpackst.

Test 112 Monaden sind ausdrucksstärker als applikative Funktoren, und applikative Funktoren sind ausdrucksstärker als Funktoren.
- Implementiere `fmap`, `pure` und `(<*>)` mithilfe von `return` und `(>>=)`.
- Implementiere `fmap` mithilfe von `pure` und `(<*>)`.

Test 113 Betrachten wir die Typen von `fmap` und `(>>=)`, dann sehen wir gewisse Ähnlichkeiten.

```
fmap :: (a -> b) -> f a -> f b
(>>=) :: m a -> (a -> m b) -> m b
```

Wie können wir bereits an den Typen sehen, dass `(>>=)` die mächtigere Funktionen der beiden ist? Beziehe in deine Überlegungen ein, dass

```
fmap f (Right x) = Left y
```

nie gelten kann.

Test 114 Argumentiere anhand der Gesetze, die für eine `Functor`-Instanzen gelten sollen, dass die folgenden `Functor`-Instanzen keine gültigen Instanzen sind. Gebe auch Beispiele an, die zeigen, dass die Gesetze nicht erfüllt sind.

```
instance Functor [] where
  fmap f [] = []
  fmap f (x:xs) = f x : f x : xs

data Tree a = Empty | Leaf a | Tree a :+: Tree a

instance Functor Tree where
  fmap _ Empty     = Empty
  fmap f (Leaf _)  = Empty
  fmap f (l :+: r) = fmap f l :+: fmap f r
```

Challenge 12 Gegeben sei der Datentyp

```
newtype ZipList a = ZipList { getZipList :: [a] }.
```
[14]

Das Ziel ist es, `ZipList` als n-stellige Generalisierung von `zipWith` zu verwenden:

```
f <$> ZipList xs1 <*> ... <*> ZipList xsN
```

- Implementiere eine `Functor`-Instanz für `ZipList`.
- Bevor du eine `Applicative`-Instanz für `ZipList` implementierst, überlege warum

  ```
  pure :: a -> ZipList a
  pure x = ZipList [x]
  ```

  keine gültige Definition ist? Welche Gesetze wären verletzt, würdest `pure` so definieren?
- Implementiere eine `Applicative`-Instanz für `ZipList`.
- Zeige, dass sowohl die `Functor`- als auch die `Applicative`-Instanz die üblichen geforderten Gesetze erfüllen.

Test 115 Wie übersetzen wir

```
do x <- e1; e2   und   do e1; e2
```

[14] Das Umwickeln eines Typen mit `newtype`, für den wir bereits Typklasseninstanzen haben, ist ein gängiger Trick, um alternative Instanzen für diese Typklassen bereitzustellen.

in einen äquivalente Ausdrucke mithilfe von `(>>=)` und `(>>)`?

Hier ist ein größerer Ausdruck:

```
do
  x <- getInt
  y <- getInt
  print (x + y)
  return (x + y)
```

Test 116 Wie übersetzen wir die Audrücke

```
eval e1 >>= \x -> eval e2
        >>= \y -> if y == 0
                    then Nothing
                    else return (x + y)
```

und

```
getLine >> read <$> getLine
        >>= \x -> case f x of
                    Nothing -> return 0
                    Just _  -> return x
```

in einen äquivalenten Ausdruck mithilfe der `do`-Notation?

Test 117 Gegeben sei der Datentyp `Tree a = Leaf a | Tree a :+: Tree a`.

- Implementiere eine Funktion `splits :: [a] -> [([a], [a])]`, die alle nicht-leeren Aufteilungen der Eingabeliste berechnet.

  Zum Beispiel soll `splits [1..4]` die Liste `([1], [2, 3, 4]), ([1, 2], [3, 4]), ([1, 2, 3], [4])]` ergeben.
- Implementiere eine Funktion `allTrees :: [a] -> [Tree a]`, die alle Binärbäume generiert, deren Blätter von links nach rechts die Eingabeliste lesen. Versuche, `allTrees` mithilfe von list comprehensions oder der Listenmonade zu implementieren.

Test 118 Gegeben ist der Typ `data Deep a b = Deep [Maybe (Either a (Deep a b))]`. Die Faltungsfunktion sieht im Wesentlichen so aus.

```
foldDeep :: ([Maybe (Either a r)] -> r) -> Deep a b -> r
foldDeep fdeep (Deep x) = fdeep (f x)
  where fold = foldDeep fdeep
```

Wie können wir `f :: [Maybe (Either a (Deep a b))] -> [Maybe (Either a r)]` definieren?

Test 119 Oft kommt es vor, dass Berechnungen zustandabhängig verschiedene Ergebnisse liefern. Zum Beispiel merken wir uns in einer Tiefensuche durch einen Graph, welche Knoten bereits besucht wurden, damit die Tiefensuche sich nicht in einem Kreis verläuft. Allgemein können wir solche Berechnungen als Funktionen vom Typ `s -> (a, s)` auffassen. Sie bekommen ein Zustand vom Typ `s` und liefern ein Ergebnis vom Typ `a` und einen (möglicherweise) neuen Zustand (ebenso vom Typ `s`).

Implementiere eine Funktion `sequence :: [s -> (a, s)] -> s -> ([a], s)`, die Liste zustandabhängiger Berechnungen nimmt und der Reihe nach ausführt. Es wird dabei ein erster Zustand übergegeben, der die erste Berechnung anstößt. Das Ergebnis soll der letzte Zustand mit allen Ergebnissen der Berechnungen sein.[15]

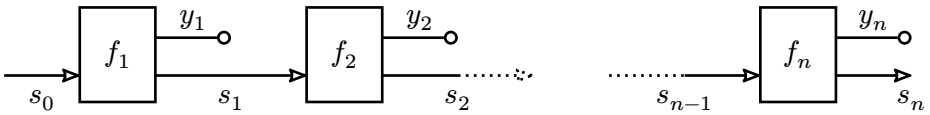
Betrachte folgendes Beispiel:

```haskell
--      Zustand        Ergebnis  neuer Zustand
fib :: (Int, Int) -> (Int     , (Int, Int)   )
fib (f0, f1) = (f0, (f1, f0 + f1))

fibs :: Int -> [Int]
fibs n = fst (sequence (replicate n fib) (0, 1))
```

Hier wird der erste Zustand mit den ersten beiden Fibonacci-Zahlen initialisiert, also $(0, 1)$. In jedem Schritt wird der Zustand mit der nächsten Fibonacci-Zahl aktualisiert und die kleinere Fibonacci-Zahl des Zustands zurückgegeben. Zuletzt projezieren wir mit `fst` auf das Ergebnis von `sequence` und verwerfen so den letzten Zustand.

Hier ist der Datenfluss von `sequence` nochmal visualisiert:



Es soll also $\text{sequence}([f_1, f_2, ..., f_n], s_0) = ([y_1, y_2, ..., y_n], s_n)$ gelten.

Challenge 13 Im Folgenden modellieren wir einen Graph als Paar (V, E) mit einer Knoten- und Kantenbewertung $w_v : V \to A$ und $w_e : E \to B$, und $E \subseteq V \times V$.

Gegeben sei der Datentyp

```haskell
data Graph a b = Graph [(Int, a)] [(Int, b, Int)].
```

Hier entspricht der erste Parameter des Datenkonstruktors w_v und der zweite w_e.

- Implementiere eine Funktion `succs :: Int -> Graph a b -> [Int]`, die zu einem gegebenen Knoten die direkten Nachfolger berechnet.
- Implementiere eine Funktion `reachable :: Int -> Int -> Graph a b -> Bool`, die entscheidet, ob zwischen zwei Knoten ein gerichteter Pfad existiert. Starte zuerst mit einfachen Graphen und betrachte immer komplexere Graphen. Wie musst du (oder kannst du) deine Implementierung anpassen?
  - Nehme zuerst an, dass ein gegebener Graph immer ein Baum ist? Wenn also ein Pfad existiert, dann ist er eindeutig.
  - Nehme jetzt an, dass ein gegebener Graph immer azyklisch ist. Das heißt, falls ein Pfad existiert, muss dieser nicht mehr eindeutig sein. Nutze deine Erkenntnisse aus Test 119, um eine Liste aller besuchten Knoten zu verwalten.
  - Was musst du in deiner Implementierung anpassen, wenn der gegebene Graph auch zyklisch sein kann?
- Je nachdem wie du `reachable` implementiert hast, könnte es denn Anschein erwecken, dass deine Lösung alle Knoten besucht. Wieso ist das nicht der Fall? Wieso passiert das nur, wenn wir uns die Liste aller besuchten Knoten anschauen? Welche Beobachtungen machst du, wenn du `reachable 0 k (yGraph k)` berechnest?

```haskell
yGraph :: Int -> Graph () ()
yGraph k = Graph [(v, ()) | v <- [0..2 * k]]
                 (let left = [(v, (), v + 1) | v <- [1..k]]
                      right = [(v, (), v + 1) | v <- [k + 1..2 * k]]
                  in (0, (), 1) : (0, (), k + 1) : (left ++ right))
```

[15] Für Interessierte: Diese Implementierung von `sequence` ist ein Spezialfall der State-Monade. Mit der Intuition, dass wir hier Berechnungen sequenzieren, sollte es nicht überraschend sein, dass `s -> (a, s)` eine Monade ist.

Hinweise zu Tests und Challenges

Hinweis zu Challenge 1 Zur Darstellung der Multimengen eignen sich sortierte Listen gut.

Hinweis zu Challenge 1 Zur Berechnung des Schnittes können zwei sortierte Listen parallel durchlaufen werden. Wenn zwei gleiche Elemente zu Beginn der Liste stehen, wird eines der Elemente zum Ergebnis hinzugefügt. Im anderen Fall überspringen wir das jeweils kleinere Element der beiden.

Hinweis zu Challenge 12 Das Identitätsgesetz wäre verletzt, wenn man `pure` wie gegeben definieren würde. Betrachte folgendes Gegenbeispiel.

```
   pure id      <*> ZipList [1, 2]
== ZipList [id] <*> ZipList [1, 2]
== ZipList [1]
/= ZipList [1, 2]
```

Es muss also dafür gesorgt sein, dass es genügend `id`s in der linken `ZipList` gibt.

Hinweis zu Challenge 13 Falls du es nicht geschafft hast, in Test 119 `sequence` zu implementieren, kannst du diese Implementierung verwenden.

```
sequence :: [s -> (a, s)] -> s -> ([a], s)
sequence []     s = ([], s)
sequence (f:fs) s = let (y, s')   = f s
                        (ys, s'') = sequence fs s'
                    in (y:ys, s'')
```

Weitere Ressourcen

Wenn du auf der Suche nach weiteren Übungsaufgaben bist, mit denen du deine Programmierkenntnisse in Prolog verbessern möchtest, bietet sich die Liste P-99: Ninety-Nine Prolog Problems an. Lösungen sind ebenso auf der Seite verfügbar. Für Haskell gibt es eine ähnliche Seite H-99: Ninety-Nine Haskell Problems.

Weitere Links:

- Learn You A Haskell
- Haskelite: Ein Schritt-für-Schritt Interpreter für (eine Teilmenge von) Haskell
- Functors, Applicatives, And Monads In Pictures
- Haskell Cheatsheet

Appendix

Bemerkung 1 In Test 68 haben wir gesehen, wie deklarative Programmierkonzepte aus Haskell in Python verwendet werden können. In Java haben diese über die letzten Jahre auch einen Platz gefunden. Hier ist das Programm in Java (ohne gecurryte Funktionen und partielle Applikation).

```
import java.util.*;
import java.util.function.*;

interface Numeric<T> {
  T zero();
  T add(T a, T b);

  public static Numeric<Integer> integer() {
    return new Numeric<>() {
      public Integer zero() { return 0; }
      public Integer add(Integer a, Integer b) { return a + b; }
    };
  }
}

interface Foldable<T> {
  <R> R foldr(R initial, BiFunction<T, R, R> function);

  default List<T> toList() {
    return foldr(new LinkedList<>(), (value, list) -> {
      list.add(value);
      return list;
    });
  }

  default int length() { return foldr(0, (_, result) -> result + 1); }

  default boolean contains(T value) {
    return foldr(false, (other, result) -> result || other.equals(value));
  }

  default T sum(Numeric<T> numeric) {
    return foldr(numeric.zero(), numeric::add);
  }
}

sealed interface Tree<T> extends Foldable<T>
  permits Empty, Node {}

record Empty<T>() implements Tree<T> {
  @Override
  public <R> R foldr(R initial, BiFunction<T, R, R> function) { return
initial; }
}

record Node<T>(Tree<T> left, T value, Tree<T> right) implements Tree<T> {
  @Override
  public <R> R foldr(R initial, BiFunction<T, R, R> function) {
    R x = right.foldr(initial, function);
```

```
    R y = function.apply(value, x);
    return left.foldr(y, function);
  }
}

public class Main {
  public static void main(String[] args) {
    Tree<Integer> tree = new Node<>(
      new Empty<>(),
      3,
      new Node<>(new Node<>(new Empty<>(), 7, new Empty<>()), 4, new Empty<>())
    );

    System.out.println(tree.sum(Numeric.integer()));  // 14
    System.out.println(tree.toList());  // [4, 7, 3]
    System.out.println(tree.length());  // 3
    System.out.println(tree.contains(3));  // true
    System.out.println(tree.contains(9));  // false
  }
}
```