Dieses Dokument ist vom 25.01.2026. Die aktuelle Version des Dokuments kannst du im moodle oder direkt von GitHub herunterladen.

Dieses Dokument enthält 249 Fragen, 23 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 die zusätzlichen Konzepte, dir zu sehr Schwierigkeiten bereiten, überspringe die entsprechende Frage oder Aufgabe.

Die Nomenklatur des Aufgaben ist aktuell möglicherweise noch etwas willkürlich, da es Tests gibt, die wie Challenges wirken – und möglicherweise sogar andersherum.

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? Warum ist die Abwesenheit von Seiteneffekten wünschenwert, sofern es möglich ist?

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 Welche Vorteile und Nachteile haben streng getypte Programmiersprachen?

Test 5 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 6 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 und gebe Beispiele an.

Test 7 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 8 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 9 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], sofern wir die Laufzeit für die Verknüpfung vernachlässigen können. 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 10 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 11 Ist der folgende Ausdruck typkorrekt?

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

Test 12 Wie werden algebraische Datentypen in Haskell definiert?

Test 13 Was ist charakterisierend für Aufzählungstypen, Verbundstypen und rekursive Datentypen? Gebe Beispiele für jeden dieser Typarten an.

Test 14 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 mit weniger Arbeit an dein Ziel kommst?

[1] Binäre Exponentiation

Test 15 Wie ist die Funktion `lengthIntList :: IntList → Int` aus dem vorherigen Test definiert?

Test 16 Du hast einen Datentypen definiert und möchtest dir Werte des Typen nun z.B. im GHCi anzeigen lassen. Was kannst du tun, um dieses Ziel zu erreichen?

Test 17 Wie definieren wir Funktionen?

Test 18 Gebe ein Listendatentypen an, für den es nicht möglich ist, kein Element zu enthalten.

Test 19 In Programmiersprachen wie Java greifen wir auf Daten komplexer Datentypen zu, indem wir auf Attribute von Objekten zugreifen oder getter-Methoden verwenden. Wie greifen wir auf Daten in Haskell zu?

Test 20 Wie sieht eine Datentypdefinition in Haskell im Allgemeinen aus?

Test 21 Welchen Typ haben

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

Test 22 Was ist parametrischer Polymorphismus?

Test 23 Welche Typkonstruktoren des kinds `* → *` kennst du?

Test 24 Welchen kinds haben jeweils `Either` und `Either a`?

Test 25 Beim Programmieren in Haskell vernachlässigen redundante Syntax. Gibt es in Haskell einen Unterschied zwischen `f 1 2` und `f(1, 2)`.

Test 26 Welches Konzept erlaubt es uns, dass wir Funktionen auf Listen nicht für jeden konkreten Typen angeben müssen?

Test 27 Wie gewinnt man aus einem Typkonstruktor einen Typ?

Test 28 Visualisiere `[1, 2, 3]` als Termbaum, 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 29 Ist `[32, True, "Hello, world!"]` ein valider Haskell-Wert? Warum ja oder nein?

Test 30 Was ist der Unterschied zwischen einem Typ und einem Typkonstruktor?

Test 31 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:

```
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] Die Funktionen müssen nach dem Ableiten nicht vereinfacht werden.

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.

Später kannst du auch hier die jeweiligen Typklassen verwenden, um die arithmetischen Operationen zu überladen.

[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 32 Wie können wir es hinkriegen, dass die invalide Liste `[32, True, "Hello, world!"]` ein valider Haskell-Wert wird? Mithilfe welches Hilfstypen kriegen das hin? (Die Liste müssen wir dafür unter Umständen umschreiben.)

Test 33 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.

Test 34 Hier ist eine fehlerhafte Implementierung eines Datentyps für einen knotenbeschrifteten Binärbäumen.

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

Was ist der Fehler?

Test 35 In imperativen Programmiersprachen (hier Java) iterieren wir über Listen oft in folgender Form.

```
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.

Dein Ergebnis sollte Haskell-idiomatisch sein.

Test 36 Die `(!!)`-Funktion ist unsicher in dem Sinne, dass sie für invalide Listenzugriffe einen Fehler wirft – also z.B. für `xs !! (-1)` oder `xs !! k` mit `k > length xs`. Die Funktion `(!?) :: [a] -> Int -> Maybe a` ist eine sichere Variante von `(!!)`. Sie macht den Fehlerfall explizit durch die Wahl des Ergebnistypen. Wie fängt der Ergebnistyp diesen Fehlerfall auf? Implementiere diese Funktion.[4]

Test 37 Gegeben sei der Datentyp für knotenbeschriftete Binärbäume

```
data Tree a = Empty | Node (Tree a) a (Tree a).
```

Mithilfe einer Pfadbeschreibung können wir durch so einen Baum navigieren. Diese Beschreibung soll durch eine Liste von Werten vom Typ `data D = L | R` dargestellt sein.[5] Implementiere eine Funktion `(!?) :: Tree a -> [D] -> Maybe a`, die die Beschriftung des

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

[5] Da Datenkonstruktoren in Haskell nicht überladen werden können, können hier leider nicht `Left` und `Right` verwendet werden, solange die Datenkonstruktoren des `Either`-Typs im scope sind.

Knotens zurückgibt, der durch die gegebene Pfadbeschreibung gefunden wird. Hier sind kleine Beispiele:

- `Node Empty 3 Empty !? [] = Just 3`
- `Node Empty 3 Empty !? [L] = Nothing`
- `Node Empty 3 Empty !? [R] = Nothing`
- `Node (Node Empty 1 (Node Empty 2 Empty)) 3 Empty !? [L, R] = 2`

Test 38 `(++) :: [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 wäre.

Test 39 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 Haskell sind Listen als einfach-verkettete Listen implementiert. Das macht sie ungeeignet für Operationen, die wahlfreien Zugriff in konstanter Laufzeit benötigen. Darüber hinaus sind Listen auch nicht mutierbar. Das führt dazu, dass Operationen, die eine Liste verändern, häufig lineare Laufzeit in der Länge der Liste haben – mit Ausnahme der `(:)`-Operation.

Ziel dieser Challenge ist es, eine Datenstruktur zu entwickeln, die eine (amortisiert) konstante `append`-Operation hat. Diese ist bekannt als `Queue`. Sie soll durch `data Queue a = Q [a] [a]` dargestellt werden. Die Idee ist es, eine (linke) Liste vorzuhalten, die eine (amortisiert) konstante `dequeue`-Operationen erlaubt, und eine andere (rechte), die eine konstante `enqueue`-Operationen erlaubt. Das heißt, die fast alle dieser Operationen benötigen konstante Laufzeit und konstant wenige können lineare Laufzeit haben.

Implementiere die Funktionen

- `empty :: Queue a`, die eine leere Queue erzeugt,
- `front :: Queue a → a`, die das erste Element in einer queue zurückgibt,
- `isEmpty :: Queue a → Bool`, die bestimmt, ob eine queue leer ist,
- `enqueue :: a → Queue a → Queue a`, die ein Element an das Ende einer queue anhängt,
- `dequeue :: Queue a → Queue a`, die das erste Element in einer queue entfernt.

Die Implementierung soll dabei folgende Invariante erfüllen: Eine queue ist genau dann leer, wenn die `dequeue`-Liste leer ist. Diese Invariante kannst du z.B. mit einer Hilfsfunktion erzwingen – oder du passt bei der Implementierung deiner Funktionen auf.[6] Falls es dir für den Anfang einfacher fällt, ignoriere die Invariante erstmal.

[6] Für Interessierte: Wenn sogar `length xs ≥ length ys` für eine queue `Q xs ys` gewährleistet wird, ist die queue nochmal schneller. Dafür muss man die Längen der Listen immer vorhalten. Mehr darüber findest du in Simple and efficient purely functional queues and deques von Chris Okasaki lesen. Falls dich funktionale Datenstrukturen allgemein interessieren, sei dir seine Doktorarbeit empfohlen.

Challenge 6 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.

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.[7]

Test 40 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 41 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 42 Wie können wir eine Funktionen teilweise auf Korrektheit testen – also wie können wir für eine beliebige Eingabe verifizieren, dass die Ausgabe korrekt ist?

Test 43 Welchen Nachteil hat die Prüfung von Vorbedingungen mit `(⟹)`? Wie können wir diese beheben?

Test 44 Was sind Funktionen höherer Ordnung?

Test 45 Wie definieren wir Lambda-Abstraktionen bzw. anonyme Funktionen?

Test 46 Warum ist der Typ `(a → b) → c` nicht identisch zum Typ `a → b → c`? Welcher andere Typ ist identisch zu letzterem?

Test 47 Mit welchen Konzepten gehen die Linksassoziativität der Funktionsapplikation und die Rechtsassoziatvität des Typkonstruktors `(→)` gut Hand in Hand?

Test 48 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 49 Was ist partielle Applikation?

[7] Rebalancierung eines AVL-Baum

Test 50 Was ist Currying?

Test 51 Welche Funktionen höherer Ordnung hast du kennengelernt im Kontext der generischen Programmierung? Was ist das Ziel dieser Funktionen?

Test 52 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 lauten diese Definitionen?

Test 53 Gegeben seien folgende Funktionen:

- `rgbToHsv :: RGB → HSV`, die eine Farbe von einer Darstellung in einen anderen konvertiert, und
- `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 54 Mit Funktionen höherer Ordnung können wir Kontrollstrukturen aus der imperativen Programmierung definieren. Hier ist eine mögliche Definition einer bedingten Wiederholung.

```
while :: (a → Bool) → (a → a) → a → a
while p f x | p x       = while p f (f x)
            | otherwise = x
```

In Test 5 haben wir zwei Funktionen gesehen, die diese Kontrollstruktur verwenden. In Haskell können wir `clz` mithilfe von `while` wie folgt definieren.

```
clz :: Int → Int
clz n = snd (while cond step (n, 0))
  where
    cond (n, k) = n > 0
    step (n, k) = (n `div` 2, k + 1)
```

Oder alternativ so

```
clz :: Int → Int
clz = go 0
  where
    go k n | n > 0     = go (k + 1) (n `div` 2)
           | otherwise = 64 - k
```

Beide Funktionsdefinitionen sind semantisch äquivalent. Argumentiere unter verschiedenen Aspekten, warum die eine Implementierung besser als die andere sein könnte – z.B. in Hinsicht auf Lesbarkeit, Idiomatik und Wartbarkeit.

Test 55 Was sind sections im Kontext von Funktionen höherer Ordnung?

Test 56 Welche der Faltungsfunktion auf Listen ergibt sich aus dem Verfahren zur Erzeugung von Faltungsfunktionen, das du für beliebige Datentypen kennengelernt hast?

Test 57 Was ist der Unterschied zwischen `foldl` und `foldr`? Wann liefern `foldl` und `foldr` das gleiche Ergebnis?

Test 58 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 59 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 60 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 61 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 62 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 63 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 64 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`.

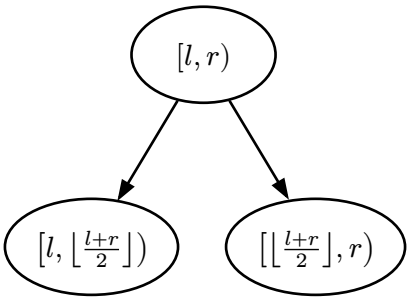
- Implementiere `zip` mithilfe von `zipWith`.
- Implementiere das Prädikat `isSorted` mithilfe von `zipWith`.

Test 65 Das Pendant zum Falten ~~ist das Bügeln~~ mit `foldr` ist `unfoldr` (aus `Data.List`). Anstatt eine Liste von Werten zu falten, können wir mit `unfoldr` aus einem Wert eine Liste erzeugen. Die Funktion hat den Typ `(b → Maybe (a, b)) → b → [a]`.

- Überlege dir anhand des Typs, wie diese Funktion implementiert sein könnte. Implementiere sie anschließend.
- Können wir `foldr (+) 0 [3, 1, 4, 1, 5]` mit `unfoldr` rückgängig machen? Das heißt, können wir die Eingabeliste rekonstruieren, ohne Annahmen darüber zu machen, wie die Eingabeliste entstanden ist?
- Berechne die Binärdarstellung einer natürlichen Zahl mithilfe von `unfoldr`. (Das LSB soll an erster Stelle der Ausgabeliste stehen.) Implementiere dies als Funktion `bits :: Int → [Int]`.
- Implementiere `map` mithilfe von `unfoldr`.

Test 66 In diesem Test wollen wir einen beliebigen Wert zu einen Baum entfalten. Implementiere eine Funktion `unfoldTree :: (b → Maybe (b, a, b)) → b → Tree a`, die eine Funktion nimmt, die ein Wert von Typ `b` nimmt und in eine Knotenbeschriftung und zwei weitere Wert vom Typ `b` aufspaltet, oder das Entfalten stoppt, indem ein `Nothing` zurückgegeben wird. Bäume sind durch den Datentyp `data Tree a = Empty | Node (Tree a) a (Tree a)` definiert.

Nutze die definierte Funktion, um einen Binärbaum zu erzeugen, welche die Zerlegungen eines diskreten Intervals darstellen soll. Entnehme die Art der Zerlegung dem Diagramm.



Challenge 7 Gegeben sei die 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)`.

Eine Reihe von Funktionen, die du bereits für Listen kennengelernt hast, lassen sich auch auf Bäume übertragen.[8] 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 67 Gegeben seien die Funktion

[8] 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.

```
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 68 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)
```

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

Test 69 Welche Funktion verbirgt sich hinter `foldr ((++) . f) []` und was ist ihr Typ?

Test 70 Versuche in den folgenden Ausdrücken, Teilausdrücke schrittweise durch bekannte Funktionen zu ersetzen oder 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 snd) x`.[9]

Challenge 8 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.

[9] „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.

- 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 71 Eta-reduziere die folgende Ausdrücke:
- `sum xs = foldr (+) 0 xs`,
- `add a b = a + b` und
- `\x ys → (:) x ys`.

Test 72 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 73 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 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 74 Gegeben sei folgendes Python-Programm.

```python
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, diese Konzepte im Python-Programm zu identifizieren.

Wo findest du
- Funktionen höherer Ordnung,
- pattern matching,
- algebraische Datentypen (Typkonstruktoren, Datenkonstruktoren),
- parametrischen Polymorphismus[10],
- 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 2 kannst du das gleiche Programm in Java sehen.

Test 75 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?

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

Test 76 Was ist ein abstrakter Datentyp? Was sind die Bestandteile eines abstrakten Datentyps?

Test 77 Wie definieren wir die Semantik der zu einem abstrakten Datentyp gehörenden Operationen? Wie definieren wir sie insbesondere nicht?

Test 78 Wieso ist das sofortige Nutzen einer Gleichheit auf einem abstrakten Datentypen problematisch? Was sollte man stattdessen tun?

Test 79 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 80 Welche Eigenschaften sollten die für einen abstrakten Datentypen formulierten Gesetze erfüllen, damit sie eine sinnvolle Semantik beschreiben?

Challenge 9 Gebe folgende abstrakte Datentypen an: Paar, Menge, stack, queue, double-ended queue, knotenbeschrifteter Binärbaum, priority queue.

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

Test 81 Als Teil eines ADTs für Arrays soll eine Operation `reverse :: Array a → Array a` spezifiziert werden, die ein Array umdreht. Ihr Verhalten soll unter anderem durch das folgende Gesetz festgehalten sein:

```
reverse (reverse a) == a
```
für alle Arrays `a`

Warum ist dieses Gesetz problematisch? Wie können wir das Problem beseitigen?

Test 82 Als Teil eines ADTs für Array soll eine Operation `at :: Array a → Int → a` spezifiziert werden, die das Element an einer Position in einem Array zurückgibt. Weiter soll `update :: Int → a → Array a → Array a` einen Wert an eine Position in ein Array schreiben.

Wie können wir spezifizieren, dass durch ein `update` nur das Element an der gegebenen Position verändert wird?

Test 83 Eine Teilmenge der Operationen für eine Menge sind
- `insert :: a → Set a → Set a` zum Einfügen von einem Wert in eine Menge und
- `size :: Set a → Int` zum Bestimmen der Kardinalität einer Menge.

Gegeben ist folgendes Gesetz:

```
size (insert y (insert x s)) == size s + 2
```
für alle Werte `x, y`, alle Mengen `s`.

Das Gesetz ist falsch. Warum und wie können es korrigieren?

Test 84 In ADT-Gesetzen sind Variablen allquantifiziert. Wie können wir gewährleisten, dass ein Wert bestimmte Bedingungen erfüllt, bevor wir ein entsprechendes Gesetz für solche Werte definieren?

Test 85 Warum benötigen wir Konstruktoren als Teil eines ADTs?

Test 86 Im Kontext der objektorientierten Programmierung hast du wahrscheinlich das Konzept der Datenkapselung kennengelernt. Dabei geht es um das Verbergen von Daten, sodass Zugriffe von außen nicht möglich sind. In der objektorientierten Programmierung wird dies z.B. durch explizite Angabe von Zugriffsarten für Attribute oder Methoden erreicht.

In Haskell haben wir so etwas `private` und `public` nicht. Wie können wir aber trotzdem verhindern, dass bestimmte Operationen auf Werten eines Datenttypen nicht möglich sind? Welche Rolle spielen smart constructors in diesem Zusammenhang?

Test 87 Was sind Typklassen?

Test 88 Wie unterscheidet sich der Polymorphismus, der durch Typklassen ermöglicht wird, vom parametrischen Polymorphismus?

Test 89 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 90 Welche Typklassen kennst du? Was ermöglichen sie konkret?

Test 91 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 Werte vom Typ `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 92 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.[11]

Test 93 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.

Challenge 10 In dieser Challenge sollst du automatisches Differenzieren im Rückwärtsmodus mithilfe von (Operator-)Überladung implementieren. Dieser Ansatz des Differenzierens führt dabei das Differenzieren komplizierter Funktionen auf einfache, elementare Funktionen zurück.

Wir verwenden folgenden Datentyp: `data D a = D a a`. Ein Wert vom Typ `D a` enthält einen Funktionswert und seine Ableitung.

Der Kern der Idee ist, Funktionen so zu überladen, dass sie auf `D a`-Werte angewendet werden können. Angenommen, es sei eine Funktion `f` und ihre Ableitungsfunktion `f'` gegeben. Dann soll ein überladenes `f` wie folgt funktionieren

```
f :: D a → D a
f (D gx dgdx) = D (f gx) (dgdx * f' gx)
```

Der Wert `gx` ist das Ergebnis einer inneren Funktion `g`, und `dgdx` entspricht deren Ableitung `g'` an der Stelle `x` (bzw. $\frac{dg}{dx}(x)$). Die Kettenregel führt dann zu `(f . g)' z = g' z * f' x`. Nach

[11] 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.

dem Muster kannst du nun Standardfunktionen überladen. Für die arithmetischen Operatoren benötigst du an der Stelle deren Ableitungsregeln (Summenregel, Leibnizregel, usw.)

Implementiere die Typklasseninstanzen `Num`, `Fractional` und `Floating`.[12]

Mit den folgenden Funktionen kannst du dann die erste, zweite oder dritte Ableitung bilden.[13]

```
d1 :: Num a ⇒ (D a → D b) → a → b
d1 f x = let (D _ d) = f (D x 1) in d

d2 :: Num a ⇒ (D (D a) → D (D b)) → a → b
d2 f x = let (D (D _ _) (D _ d)) = f (D (D x 1) 1) in d

d3 :: Num a ⇒ (D (D (D a)) → D (D (D b))) → a → b
d3 f x = let (D _ (D _ (D _ d))) = f (D (D (D x 1) 1) 1) in d
```

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

Test 95 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?
- 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 96 Welche Funktionen musst du implementieren, damit eine `Ord`-Instanz vollständig definiert ist? Welche Gesetze sollten die Funktionen einer `Ord`-Instanz erfüllen?

Test 97 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?

[12] Zusammenfassung der Ableitungsregeln

[13] Für Interessierte: In Bemerkung 1 kannst du eine allgemeinere Funktion zum Berechnen der Ableitung sehen.

- 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 98 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.[14] Woran liegt das? Mit Hinsicht auf Effizienz – welche der beiden Funktionen würdest du implementieren, wenn du nur eine implementieren dürftest?[15]

Test 99 In nicht streng getypten Programmiersprachen haben wir oft mit impliziter Typkonversion zu tun.[16] 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 100 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`, die diese Strukturen implementieren. Gebe auch beispielhaft ein paar Instanzen für diese an.

Test 101 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 102 Was ist Lazy Evaluation?

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

[15] 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.

[16] Diese wollen wir nun für einen Moment nach Haskell zurückholen, um sie dann ganz schnell wieder zu vergessen.

Test 103 Wie werden Berechnungen in Haskell angestoßen? Wie viel wird berechnet?

Test 104 Gebe ein Beispiel an, das zeigt, dass die faule Auswertung berechnungsstärker ist.

Test 105 Welche praktischen Vorteile ergeben sich aus der Lazy Evaluation?

Test 106 Wie werden mehrfache Berechnungen in einer nicht-strikten Auswertungsstrategie vermieden?

Test 107 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 108 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 korrespondiert diese verkettete Liste mit der unendlichen Liste `[1, 2, 1, 2, ...]`.

Die Mutierbarkeit des `next`-Zeigers macht das Verlinken der Knoten in Python möglich. Wie können wir in Haskell, trotz der Abwesenheit von Mutierbarkeit, zyklische Datenstrukturen umsetzen? Versuche, dein Programm ähnlich zum Python-Programm aussehen zu lassen.[17]

Challenge 11 Gegeben sei der Datentyp

```
data Tree a = Empty | Node (Tree a) a (Tree a).
```

- Implementiere eine Funktion `preorder :: Tree a → [a]`, die Knotenwerte in pre-order zurückgibt. Das heißt, zuerst wird ein Knoten betrachtet und anschließend dessen linker und danach dessen rechter Teilbaum.
- Implementiere einen unendlichen Baum `tree :: Tree Int`, der die Menge $\{f(i,j) \mid i,j \in \mathbb{N}, i \leq j\}$ mit $f(i,j) = i + j + 2ij$ darstellt. Die Wurzel soll den Wert $f(1,1)$ haben. Für einen beliebigen Knoten mit Beschriftung $f(i,j)$ soll die Wurzel des linken Teilbaums mit $f(i+1,j)$ beschriftet sein und die Wurzel des rechten Teilbaums mit $f(i,j+1)$. Falls $i > j$ erreicht wird, soll in den Baum ein `Empty`-Knoten gesetzt werden.
- Wende `preorder` auf `tree` an. Welches Problem haben wir hinsichtlich der Werte, die wir den jeweiligen Teilbäumen von `tree` sehen und der Ergbenisliste von `preorder`? Wie hängt deine Beobachtung mit `[f i j | i ← [1..], j ← [i..]]` zusammen?
- Implementiere eine Abwandlung von `preorder`, die statt `(++)` die Funktion `merge` verwendet.
- Implementiere als Nächstes die Mengendifferenz als Funktion `diff :: Ord a ⇒ [a] → [a] → [a]`. Du darfst dabei annehmen, dass die Eingabelisten bereits sortiert sind.
- Was berechnet `2 : map (\x → 2 * x + 1) ([1..] `diff` preorder tree)`?[18]

[17] Für Interessierte: Die Technik ist als Tying the Knot bekannt.

Test 109 Gegeben sei der Datentyp

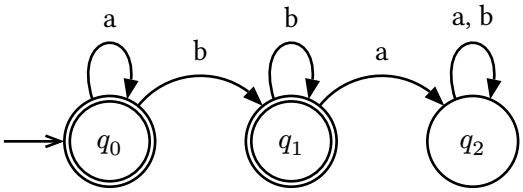
```
data Doubly a = Null | Node (Doubly a) a (Doubly a).
```

- 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 muss es nicht möglich sein, zum anderseitig verketteten Element zurückzukommen. `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?

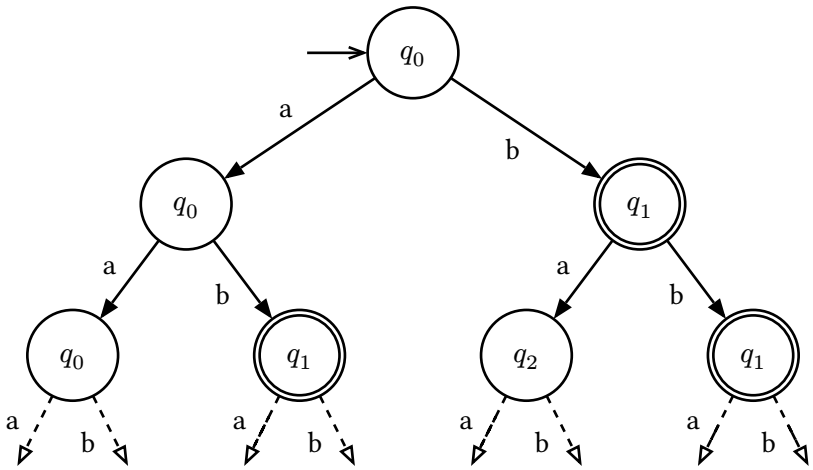
Der Wert des folgenden Ausdrucks soll `8` sein.

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

Challenge 12 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 als `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.

[18] Das Verfahren ist als Sieb von Sundaram bekannt. Die Konstruktion der oben angegebenen Menge mithilfe von unendlichen Bäumen ist nicht Teil des Verfahrens – sondern Willkür des Verfassers.

- 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 110 Wieso können wir mit `foldl` auf unendlichen Listen mit keinem Ergebnis rechnen?

Test 111 `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 13 Fixpunktverfahren sind iterative Methoden, bei denen eine Funktion wiederholt auf einen Wert angewendet wird, bis sich ein stabiler Punkt (ein sogenannter Fixpunkt) ergibt, der sich durch weitere Anwendungen nicht mehr verändert.

Dieses Berechnungsmuster wird durch die Funktion `iterate :: (a -> a) -> a -> [a]` in der Prelude festgehalten. Sie berechnet eine unendliche Liste, bestehend aus den Ergebnissen der wiederholten Anwendungen der übergegebenen Funktion auf den gegebenen Wert. Das erste Ergebnis ist der gegebene Wert, auf den die Funktion noch nicht angewendet wurde.

- Implementiere `iterate`.
- Ein klassisches Beispiel für ein Fixpunktverfahren 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`.
- Mithilfe des Differenzenquotienten kannst du die erste Ableitung approximieren (oder mit der Lösung von Challenge 2 oder Challenge 10). Diese benötigt man unter anderem für das Gradientenverfahren. Eine vereinfachte Iterationsvorschrift des Verfahren ist gegeben durch

$$x_{k+1} = x_k - 10^{-4} f'(x_k) \quad \text{für alle } k \in \mathbb{N}$$

  und einem Startpunkt $x_0 \in \mathbb{R}$. Implementiere das Verfahren mithilfe von `iterate`. Wähle als Ergebnis die erste Nährung x_{n+1}, die $\left|x_{n+1} - x_n\right| < \varepsilon$ für ein festes $\varepsilon > 0$ erfüllt (z. B. $\varepsilon = 10^{-5}$).

Challenge 14 Eine Editierdistanz zwischen zwei Wörtern $u \in \Sigma^m, v \in \Sigma^n$ können wir mithilfe der folgenden Rekurrenz bestimmen:

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

für alle $0 \leq i \leq m, 0 \leq j \leq n$. Um $\operatorname{ed}(m,n)$ 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 und ohne die Optimalität des Ergebnisses zu gefährden.

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]]
```

Die Berechnungen sind in `table` gespeichert.

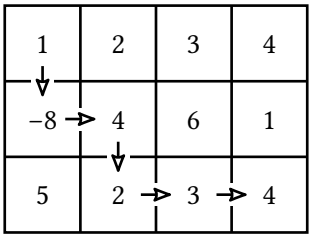
- Definiere die Funktion `ed :: Int → Int → Int` lokal in `editdist`. `ed` soll hier die zwischengespeicherten Ergebnisse aus `table` verwenden und dadurch die Berechnungen anstoßen.
- Überlege dir wie hier laziness und memoization zusammenspielen.
- Welche worst-case Laufzeit hat deine Lösung, wenn du annimmst, dass die Laufzeit von `(!!)` konstant ist?

Challenge 15 Bevor du dich dieser Challenge stellst, bietet es sich an, sich Challenge 14 anzunehmen, da in dieser der technische Teil der Lösungsidee vorgestellt wird.

Gegeben sei ein Gitter $G \in \mathbb{Z}^{m \times n}$. Ein Pfad durch das Gitter startet oben links und endet unten rechts. In jedem Schritt kannst du von einer Zelle in die rechtsanschließende oder

darunteranschließende Zelle gehen. Die Pfadsumme ist die Summe aller Zellenwerte, durch die der Pfad führt.

Hier ist ein Beispiel für ein solches Gitter. Der Pfad der minimalen Pfadsumme ist durch die Pfeile angedeutet. Für dieses Beispiel ist die minimale Pfadsumme 6.



Implementiere eine Funktion `pathsum :: (Num a, Ord a) ⇒ [[a]] → a`, die die minimale Pfadsumme berechnet.

Test 112 Gegeben sei der Datentyp

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

- Implementiere eine `Enum`-Instanz für `Direction`, die sich wie die vom GHC abgeleitete Instanz verhält.
- Implementiere eine `Bounded`-Instanz für `Direction`, die sich wie die vom GHC abgeleitete Instanz verhält.
- 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 113 Implementiere Funktion `cycleFrom :: (Enum a, Bounded a) ⇒ a → [a]`, die ab einem gegebenen Wert alle Werte des Typen nicht absteigend durchläuft. Wenn der größte Wert erreicht ist, soll die Liste wieder beim kleinsten Wert des Typen beginnen.

Test 114 An welches mathematische Konzept sind list comprehensions angelehnt?

Test 115 Aus welchen Teilen besteht eine list comprehension?

Test 116 Implementiere die Funktionen `map`, `filter` und `concatMap` mithilfe von list comprehensions.

Test 117 Was ist referenzielle Transparenz?

Test 118 Welche Rolle spielt der Typ `IO a` bzgl. Seiteneffekte? Was beschreibt ein Wert vom Typ `IO a`?

Test 119 Wie können wir zwei `IO`-Aktionen zu einer neuen `IO`-Aktion kombinieren?

Test 120 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 121 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 122 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 123 Wie lauten die `Functor`-Gesetze?

Test 124 Die Funktor-Typkonstruktorklasse ist wie folgt definiert.

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

Welchen kind hat `f`?

Test 125 Wie sind die `Functor`-Instanzen für `Identity`, `Maybe`, `Either e`, `[]` und `((→) r)` definiert?

Test 126 Wie lauten die `Applicative`-Gesetze?

Test 127 Wie sind die `Applicative`-Instanzen für `Identity`, `Maybe`, `Either e`, `[]` und `((→) r)` definiert?

Test 128 Wie lauten die `Monad`-Gesetze?

Test 129 Wie sind die `Monad`-Instanzen für `Identity`, `Maybe`, `Either e`, `[]` und `((→) r)` definiert?

Test 130 Die `Applicative`-Typkonstruktorklasse erlaubt es uns, `fmap` auf Funktionen mit mehreren Argumenten zu verallgemeinern. Dadurch können wir

```
(+) <$> 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 131 Um zu verifizieren, dass die `Functor`-Gesetze für z.B. den Typ `Maybe a` gelten, müssen wir

- das Identitätsgesetz `fmap id = id` und
- das Kompositionsgesetz `fmap f . fmap g = fmap (f . g)`

zeigen. Wie gehen wir konkret für den gegebenen Typ vor? Wie zeigen wir die Gleichheit von Funktionen? Wenn die Gesetze nun für den Listendatentypen `[a]` zeigen wollen, was ändert sich an deinem Vorgehen?

Test 132 Warum gilt

```
(1 +) <$> Just 1 == Just (1 +) <*> Just 1 == pure (1 +) <*> Just 1?
```

Wie ergibt sich aus deinen Beobachtungen eine Definition für `fmap`? Wie können wir die Berechnung für beliebige applikative Funktoren verallgemeinern? Wie wird dann festgelegt, was während der Berechnung tatsächlich passiert?

Test 133 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 134 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. Kann `fmap`, die Struktur in Abhängigkeit des Wertes vom Typ `a` verändern?

Test 135 Gegeben sei diese fehlerhafe Definition einer sicheren Division:

```
safeDiv :: Int → Int → Maybe Int
safeDiv x y = pure (div x) <*> pure y
```

Wie können wir diese Implementierung reparieren, ohne `(>>=)` zu verwenden und die rechte Seite soweit wie möglich zu erhalten? Wie siehst du an diesem Beispiel, dass Monaden ausdrucksstärker als applikative Funktoren sind?

Test 136 Monaden bieten uns die Möglichkeit, Berechnungen als Folge von kleineren Berechnungen zu sequenzieren. Je nachdem welche Monade wir betrachten, beobachten wir verschiedene Effekte.

- Welche Monade drückt eine Berechnung aus, die kein oder genau ein Ergebnis liefern kann?
- Welche Monade drückt eine Berechnung aus, die kein Ergebnis oder beliebig viele Ergebnisse liefern kann?
- Welche Monade drückt eine Berechnung aus, die entweder fehlschlägt oder erfolgreich ist? Welche kann im Fall eines Fehlschlag zusätzliche Information hervorbringen?

Test 137 Wir nutzen Monaden zur Sequenzierung von Berechnungen. Identifiziere diese Sequenzierung in den folgenden `Monad`-Instanzen bzw. stelle fest, dass `m` in `m >>= k` zuerst berechnet wird, bevor die Berechnung mit `k` fortgesetzt wird.

```
instance Monad Maybe where
  return x = Just x

  Nothing >>= _ = Nothing
  Just x  >>= k = k x

instance Monad [] where
```

```
  return x = [x]

  []     >>= _ = []
  (x:xs) >>= k = k x ++ (xs >>= k)
```

Test 138 Die `Reader`-Monade ermöglicht es, eine gemeinsame Umgebung mit vielen Berechnungen zu teilen.

```
newtype Reader r a = Reader { runReader :: r → a }

instance Monad (Reader r) where
  return y = Reader (\_ → y)

  r >>= k = Reader (\s → let x = runReader r s
                             y = runReader (k y) s
                         in y)
```

Woran kannst du erkennen, dass die Berechnung `r` vor dessen Weiterführung mit `k` und dem Ergebnis `r` stattfindet?

Test 139 Das Pendant zur `Reader`-Monade aus Test 138 ist die `Writer`-Monade. Eine Spezialisierung der Monade soll hier die `ListWriter`-Monade sein. Diese Monade kann genutzt werden, um Zwischenergebnisse oder Logging-Informationen einer Berechnung zu speichern.

```
newtype ListWriter w a = ListWriter { runListWriter :: (a, [w]) }
```

Implementiere diese Monade. Mache dir insbesondere Gedanken darüber, wie du Berechnungen dieses Typens sequenzierst und wie du die Zwischenergebnisse von zwei Berechnungen kombinierst.

Test 140 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 16 Gegeben sei der Datentyp

```
newtype ZipList a = ZipList { getZipList :: [a] }.[19]
```

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]
```

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

keine gültige Definition ist? Welche Gesetze wären verletzt, würde man `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 141 `guard :: MonadZero m ⇒ Bool → m ()` kann genutzt werden, um eine Berechnung bedingt fehlschlagen zu lassen.[20] Zum Beispiel können wir mithilfe von `guard` eine sichere Division definieren.

```
safeDiv :: (Integral a, MonadZero m) ⇒ a → a → m a
safeDiv a b = guard (b /= 0) >> return (a `div` b)
```

- Implementiere die Funktion `guard :: Bool → Maybe ()` mit `Maybe ()` als konkreten Ergebnistypen.
- Die Typklasse `MonadZero` wird gängigerweise wie folgt definiert.

  ```
  class Monad m ⇒ MonadZero m where
    mzero :: m a
  ```

  Implementiere `guard` nun allgemein.
- Berechne `1 `safeDiv` 0 :: m Int` für `Maybe` und `[]`. Bevor das möglich ist, benötigst du entsprechende `MonadZero`-Instanzen.

Die Typklasse `MonadZero` wird so genannt, weil sie eine „monadische Null“ definiert. Bzgl. der Operation `(>>=)` verhält sich `mzero` absorbierend. Für Instanzen dieser Typklasse sollen diese Gesetze gelten.

```
mzero >>= f     = mzero
m     >>= mzero = mzero
```

Test 142 In Test 141 haben wir eine Typkonstruktorklasse definiert, die es uns erlaubt hat, einen Fehlschlag bzw. die Abwesenheit eines Ergebnisses auszudrücken. Diese können wir auch eine Abstraktionsebene früher einführen. Definiere eine weitere Typkonstruktorklasse `AlternativeZero` auf Ebene der applikativen Funktoren mit einer Funktion `empty :: f a`.

Folgende Ausdrücken sollen die gegebenen Werte haben:
- `guard (y /= 0) *> pure y = pure y` für `y /= 0` und
- `guard (y /= 0) *> pure y = empty` für `y == 0`.

`(*>) :: Applicative f ⇒ f a → f b → f b` ist ein Kombinator, der sich wie `(>>)` für applikative Funktoren verhält.

Test 143 Als motivierendes Beispiel für Monaden hast du die Auswertung eines arithmetischen Ausdrucks, gegeben als Termstruktur, kennengelernt. Dort haben wir die `Maybe` -Monade verwendet, um fehlschlagende Berechnung aufzufangen. Der Typ für die arithmetischen Ausdrücke ist gegeben durch:

```
data Exp a = Num a
           | Exp a :+: Exp a
           | Exp a :-: Exp a
           | Exp a :*: Exp a
           | Exp a :/: Exp a
```

- Implementiere `eval :: Exp Int → Maybe Int` mit dem `Maybe`-Applicative. Warum ist die Division, ohne eine weitere Hilfsfunktion nicht möglich? Was müsste diese Hilfsfunktion tun, damit die Regel für die Division funktioniert?[21]

[20] Für Interessierte: `guard` ist auf Basis von `Alternative` bzw. `MonadPlus` implementiert. `MonadZero` ist nicht Teil der Standardbibliothek, aber es ist definiert als Teil von `MonadPlus`.

[21] Der applikativen Funktor wird dadurch nicht ausdrucksstärker! Das, was der applikative Funktor nicht leisten kann, wird in die Hilfsfunktion ausgelagert.

- Implementiere `eval :: Exp Int → Maybe Int` mit der `Maybe`-Monade. Benötigst du hier die Hilfsfunktion aus der vorherigen Teilaufgabe? Warum ja oder nein?
- Wie kannst du mit Test 141 `eval` zu einer Funktion `eval :: MonadZero m ⇒ Exp Int → m Int` verallgemeinern?
- `MonadZero` erlaubt es uns, einen Fehlschlag allgemein auszudrücken. Allerdings können wir anhand des Fehlschlags alleine nicht feststellen, warum es zum Fehlschag kam. Verallgemeinere die Typkonstruktorklasse `MonadZero` zu einer Typkonstruktorklasse `MonadFail`, die es erlaubt, eine Beschreibung des Fehlschags anzugeben. Sie soll also eine Funktion `fail :: String → m a` definieren. Verallgemeinere `eval` erneut mit der neuen Typkonstruktorklasse. Gebe zusätzlich eine Instanz für den Typ `Either String` an – alternativ, implementiere eine `Functor`-, `Applicative`-, `Monad`- und `MonadFail`-Instanz für den Typen `data Result a = Failure String | Success a`.

Test 144 Wie hängen die Listenmonade und list comprehensions zusammen?

- Schreibe den folgenden Ausdruck

```
do
  x ← [1..10]
  y ← [1..10]
  guard (x + y == 10)
  return (x, y)
```

  mithilfe von list comprehensions. Der Wert des ausgerechneten Ausdrucks ist

```
[(0, 10), (1, 9), (2, 8), (3, 7), (4, 6), (5, 5),
 (6, 4), (7, 3), (8, 2), (9, 1), (10, 0)]
```

  Anhand des Ergebnisses kannst du dir die Semantik von `guard` herleiten (oder schaust dir vorher Test 141 an).
- Schreibe den folgenden Ausdruck

```
[f | n ← [0..], let f = fib n, f `mod` 2 == 0]
```

  mithilfe der Listenmonade.

Test 145 Die `do`-Notation erlaubt es uns, statements der Form `p ← e` zu schreiben, wobei `p` ein beliebiges Muster sein kann. Zum Beispiel ist folgender Ausdruck valide und berechnenbar.

```
do
  Just x ← [Just 1, Nothing, Just 2]
  return x
```

Der Wert dieses Ausdrucks is `[1, 2]`.

- In welches Problem läufst du, wenn du diesen Ausdruck mithilfe von `(>>=)` und `(>>)` ausdrücken möchtest?
- Wie kannst du das Problem beheben in diesem konkreten Fall beheben?
- Fällt dir möglicherweise ein allgemeine Übersetzungsvorschrift für statements dieser Form unter der Verwendung von `MonadFail` aus Test 143 ein?

Test 146 Wie übersetzen wir

`do x ← e1; e2` und `do e1; e2`

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

Wie übersetzen wir diesen etwas größeren Ausdruck?

```
do
  x ← getInt
```

```
  y ← getInt
  print (x + y)
  return (x + y)
```

Test 147 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 148 Gegeben sei folgendes Haskell-Programm.

```
main :: IO ()
main = do
  putStr "Hello"
  return ()
  putStrLn ", world!"
  return ()
```

Was ist die Ausgabe des Programm und warum?

Test 149 Gegeben sei folgendes fehlerhafte Haskell-Programm.

```
main :: IO ()
main = do
  q ← read <$> getLine
  if q == 0
    then main
    else
      p ← read <$> getLine
      print (p `div` q)
```

Warum ist das Programm fehlerhaft? Um den Fehler zu identifizieren, klammere alle validen Haskell-Teilausdrücke. (Du kannst davon ausgehen, dass alle Eingaben valide sind und deshalb keine weitere Fehlerbehandlung der Eingaben stattfinden muss.)

Test 150 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 151 Gegeben ist der Typ `data Deep a b = Deep [Maybe (Either a (Deep a b))]`. Die Faltungsfunktion sieht im Wesentlichen so aus.

```haskell
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 152 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.[22]

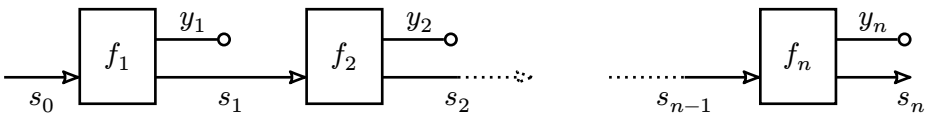
Betrachte folgendes Beispiel:

```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` das Ergebnis von `sequence` und verwerfen so den letzten Zustand.

Hier ist der Datenfluss von `sequence` nochmal visualisiert:



Es soll also `sequence [f1, f2, ... , fn] s0 = ([y1, y2, ... , yn], sn)` gelten.

Challenge 17 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.

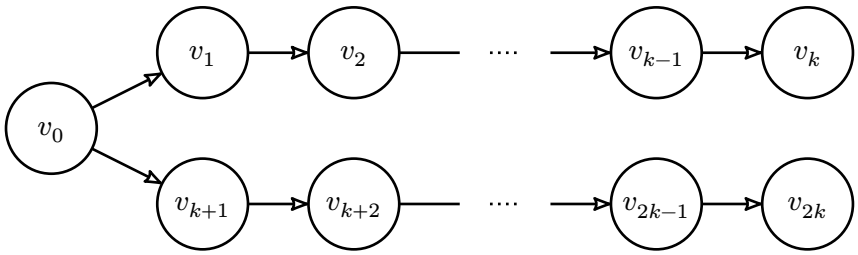
- Implementiere eine Funktion `succs :: Int → Graph a b → [Int]`, die zu einem gegebenen Knoten die direkten Nachfolger berechnet.
- Implementiere eine Funktion `reachable :: Int → Int → Graph a b → Bool`, die entscheidet, ob zwischen zwei Knoten ein gerichteter Pfad existiert. Starte zuerst mit

[22] Für Interessierte: Diese Implementierung von `sequence` ist ein Spezialfall für die State-Monade. Mit der Intuition, dass wir hier Berechnungen sequenzieren, sollte es nicht überraschend sein, dass `s → (a, s)` eine Monade ist. Alternativ kannst du die `State`-Monade implementieren und das vorimplementierte (oder von dir implementierte) `sequence` verwenden.

einfachen Graphen und betrachte immer komplexer werdende 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 152, 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 unbedingt 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?

```
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))
```

Hier ist der Graph visualiziert.



Test 153 Es kommt häufiger vor, dass zufällig generierte Werte bestimmte Eigenschaften erfüllen sollen. QuickCheck bietet auf Generatoren-Ebene z.B. die Funktion `suchThat :: Gen a → (a → Bool) → Gen a` dafür an. Diese nimmt einen Generator und generiert solange neue Werte, bis eine gewünschte Eigenschaft erfüllt ist. Diese Eigenschaft wird durch ein Prädikat formuliert.

Implementiere `suchThat`.

Es gibt ein paar QuickCheck-spezifische Fallstricke, die mit dem `size`-Parameter der Generatoren zu tun haben, die du bei deiner Implementierung nicht beachten brauchst. Falls es dich interessiert, ist hier die `suchThat`-Implementierung von QuickCheck.

Test 154 Wenn du folgende (etwas künstliche) Eigenschaft mithilfe von QuickCheck prüfst, muss im Durchschnitt jeder zweite Test verworfen werden, weil die Vorbedingung der Eigenschaft nicht erfüllt ist.

```
prop_prop :: Int → Property
prop_prop k = k > 0 ⟹ True
```

Um dieses Verhalten zu unterbinden, können wir die Eigenschaft leicht abändern.[23]

```
prop_prop :: Positive Int → Property
prop_prop (Positive k) = k > 0 ⟹ True
```

Jetzt werden keine Tests verworfen.

[23] QuickCheck hat viele solche type-level modifiers, die das Generatoren-Verhalten verändern.

Überlege dir, wie die Lösung des Problems funktioniert und implementiere sie.

Challenge 18 Eigenschaften lassen sich mit QuickCheck z.B. mithilfe der Funktion `quickCheck` prüfen.

Hier sind ein paar Eigenschaften.

```
prop_comm :: Int → Int → Bool
prop_comm x y = x + y == y + x

prop_inv :: Int → Bool
prop_inv x = x + (-x) = 0
```

- Welchen Typ muss `quickCheck` scheinbar haben, damit wir `prop_comm` prüfen können? Wie steht das im Konflikt mit `prop_inv`?
- Um `quickCheck` zu implementieren, benötigen wir die Möglichkeit, beliebig stellige Funktionen anwenden zu können. Das ist mithilfe von Typklassen möglich. Hier ist eine vereinfachte Variante der Typklasse `Testable`, wie sie in QuickCheck zu finden ist.

  ```
  newtype Property = Property { unProperty :: Bool }

  class Testable a where
    property :: a → Property
  ```

  - Implementiere `Testable`-Instanzen für `Bool` und `Property`, und anschließend eine Instanz für den Typ `a → b`, wobei `a` eine `Arbitrary`-Instanz haben soll und `b` wieder `Testable` sein soll.
  - Welche Instanzen übernehmen die Rolle eines Basisfalls und welche Instanzen übernehmen die Rolle einer induktiven Regel?
  - Wie ist es möglich, dass wir Eigenschaften definieren können, die entweder ein `Bool` oder eine `Property` als Rückgabewert haben?
  - Als Nächstes betrachten wir eine Vereinfachung der `Arbitrary`-Typklasse und eine Instanz für den Typ `Int`, damit wir `quickCheck` implementieren und verwenden können.

    ```
    class Arbitrary a where
      arbitrary :: a

    instance Arbitrary Int where
      arbitrary = 42  -- Hier werden normalerweise zufällige Werte generiert.
    ```

    Implementiere eine Funktion `quickCheck :: Testable a ⇒ a → Bool`, die eine Eigenschaft nimmt und prüft.

In QuickCheck werden Generatoren genutzt, um zufällige Werte zu generieren. Diese nutzen alle einen Zufallszahlengenerator, der als Zustand mit weiteren Parametern durch alle `arbitrary`-Aufrufe durchgetragen wird. Das haben wir uns durch die Vereinfachungen vernachlässigt.

Test 155 Implementiere Arrays als nicht-mutierende Datenstruktur in Python. Die Klasse soll so funktionieren, dass der folgende Beispielcode lauffähig ist und die angegebene Ausgabe erzeugt.

```
a = Array.fromList(5, [(2, 4), (3, 3)])

b = a.update([(1, 3), (2, 2)])
for i in range(len(a)):
  print(a[i], b[i])
```

Die Ausgabe des Programms soll

```
None None
None 3
4 2
3 3
None None
```

sein.

Challenge 19 Du hörst parallel zu diesem Modul „Berechnung und Logik“ und möchtest die Automaten-Konstruktionen für reguläre Sprachen in Haskell implementieren? Dann schaue nicht weiter. Du hast die korrekte Challenge gefunden. Dieser Weg ist lang und das erste Mal etwas steinig, also schnappe dir genügend Proviant und bringe genügend Zeit mit.

In dieser Challenge wollen wir folgende Konstruktionen implementieren.

$$\text{Regulärer Ausdruck} \xrightarrow[\text{Konstruktion}]{\text{Thompson-}} \varepsilon\text{-NFA} \xrightarrow{\varepsilon\text{-Hüllen}} \text{NFA} \xrightarrow[\text{konstruktion}]{\text{Potenzmengen-}} \text{DFA}$$

- Definiere einen Datentypen `RE`, um regulärere Ausdrücke darzustellen. Der reguläre Ausdruck `(ab)*|c` könnte z.B. so dargestellt werden

  ```
  Kleene (Literal 'a' :*: Literal 'b') :|: Literal 'c'.
  ```

  Die leere Sprache und die Sprache, die nur das leere Wort enthält, sind über das Beispiel nicht abdeckt.
- Bevor wir mit den Konstruktionen beginnen, müssen wir uns überlegen, wie wir die Automaten darstellen wollen. Überlege dir Typen für nichtdeterministische endliche Automaten mit und ohne ε-Transitionen und deterministische endliche Automaten. Einen Zustand kannst du z.B. als Ganzzahl darstellen.
- Implementiere die Thompson-Konstruktion als Funktion `RE → EpsNFA`. Hier ist eine wesentliche Schwierigkeit, neue Zustände zu erzeugen, da die Bezeichnungen verschiedener Teilautomaten miteinander kollidieren können. Das Problem lässt sich z.B. mithilfe eines Zählers lösen, der durch die Konstruktion durchgeführt wird. Eine entsprechende Hilfsfunktion könnte folgenden Typ haben: `RE → Int → (EpsNFA, Int)`. Wenn du dich an die `State`-Monade wagen möchtest, könnte dies an dieser Stelle auch nützlich sein, um den Zähler mitzuführen – im Wesentlichen versteckst du damit das explizite Mitführen des Zählers.
- Implementiere eine Funktion `epsilonClosure :: EpsNFA → [Int] → [Int]`, die die ε-Hülle einer Menge von Zuständen berechnet.
- Implementiere eine Funktion `removeEpsilon :: EpsNFA → NFA`, die alle ε-Transitionen des übergebenen nichtdeterministischen endlichen Automaten entfernt. Hier brauchst du dir keine Gedanken über neue Bezeichner für Zustände machen, da du die Zustände des alten Automaten wiederverwenden kannst.
- Implementiere die Potenzmengenkonstruktion. Dafür bietet es sich an den NEA mit einer Tiefensuche zu durchlaufen, anstatt tabellarisch jede Teilmenge der Zustandsmenge entsprechend zu verbinden. So werden dann nur Zustände durchlaufen, die für den DEA wichtig sind. Hier ist eine wesentliche Schwierigkeit, die neuen Zustände zu benennen. Es bietet sich an, ein Wörterbuch der Form `[([Int], Int)]` zu verwalten, in dem die neuen Bezeichner nachgeschaut werden können.
- Zuletzt, um zu überprüfen, ob der DEA die korrekte Sprache erkennt, implementiere eine Funktion `member :: String → DFA → Bool`, die überprüft, ob ein Wort vom übergebenen Automaten erkannt wird.

Wenn du bis hierhin gekommen bist, könntest du weiter den DEA mithilfe von Hopcrofts Algorithmus minimieren.

Logische Programmierung

Test 156 Was ist das Berechnungsprinzip von Prolog bzw. wie leitet Prolog aus gegebenen Informationen ab?

Test 157 Formuliere folgende Aussage als Prolog-Programm: Seien A und B. Dann gilt C, wenn A und B gelten.

Test 158 Wie können wir Funktionen, wie wir sie aus Haskell kennen, in Prolog umsetzen? Erkläre es mithilfe eines Beispiels (wie z.B. `(++) :: [a] → [a] → [a]`).

Test 159 n-stellige Funktionen können wir in Prolog als $(n+1)$-stellige Relation umsetzen. Dabei nimmt die letzte Position der Relation die Rolle des Ergebnisses ein. Gehen wir mit Funktionen vom Typ `a1 → ... an → Bool` besonders um?

Test 160 Wie hängen die Syntax von Prolog und die der Aussagenlogik zusammen? Welche Symbole entsprechen sich?

Test 161 In Prolog können wir Termstrukturen erzeugen. Der Haskell-Typ

```
data Maybe a = Nothing | Just a
```

kann z.B. wie folgt in Prolog umgesetzt werden.

```
maybe(nothing).
maybe(just(_)).
```

Übersetze mit der gleichen Idee den Datentyp

```
data Tree a = Empty | Node (Tree a) a (Tree a).
```

Test 162 Wie viele Ergebnisse liefern die folgenden Anfragen?

- `?- append(_, [X|_], [1, 2, 3, 4]).`
- `?- append(_, [_,X|_], [1, 2, 3, 4]).`
- `?- member(X, [1, 2, 3]), member(Y, [2, 3, 4]), X \= Y.`
- `?- append(_, [X|Ys], [1, 2, 3, 4]), append(_, [Y|_], Ys).`

Test 163 Ordne die Begriffe Atom, Fakt, Regel und Variable dem folgenden Prolog-Programm zu?

```
semitone(c, cis). semitone(cis, d). semitone(d, dis). semitone(dis, e).
semitone(e, f). semitone(f, fis). semitone(fis, g). semitone(g, gis).
semitone(gis, a). semitone(a, ais). semitone(ais, b). semitone(b, c).

minor_third(P1, Mi3) :-
  semitone(P1, Mi2), semitone(Mi2, Ma2), semitone(Ma2, Mi3).

major_third(P1, Ma3) :- minor_third(P1, Mi3), semitone(Mi3, Ma3).

chord(minor(P1, Mi3, P5)) :- minor_third(P1, Mi3), major_third(Mi3, P5).
chord(major(P1, Ma3, P5)) :- major_third(P1, Ma3), minor_third(Ma3, P5).
```

Test 164 In Test 163 sind die Halbtonschritte als Fakten definiert. Um diese etwas kompakter zu schreiben, können wir `append/3` verwenden. Implementiere `semitone/2` mithilfe von `append/3`.

Test 165 Warum ergibt es Sinn, beim Prolog-Programmieren in Relationen statt Funktionen zu denken? Betrachte z.B. das Prädikat `append/3` gemeinsam mit den Anfragen

- `?- append(Xs, Ys, [1, 2, 3]).`,

- `?- append(Xs, [2, 3], Zs).`,
- `?- append([1, 2], Xs, Zs).` und
- `?- append([1, 2], Ys, [1, 2, 3]).`.

Test 166 Wenn wir Haskell um die Möglichkeit erweitern könnten, mehrere Regeln auszuprobieren und mehrere Lösungen zu kombinieren, hätten wir trotz der Abwesenheit logischer Variablen bereits viele Möglichkeiten der Modellierung, wie wir sie in Prolog haben. Es stellt sich heraus, dass die Listenmonade den Nichtdeterminismus schon sehr gut abbilden kann, und erlaubt damit einen weiteren abstrakten Blick auf die Listenmonade - anstatt z.B. der Blick der imperativen Programmierung als Verschachtelung von Schleifen.

Als kleines Beispiel wollen wir einen fairen Münzenwurf modellieren. Wir kodieren die Ereignisse binär, wobei 0 für Kopf und 1 für Zahl stehen soll. Weiter ist auch ein Beispiel angegeben für zwei unabhängige Münzenwürfe.

In Prolog:

```
coin(0).
coin(1).

coin2(X, Y) :-
  coin(X),
  coin(Y).
```

In Haskell:

```
coin :: [Int]
coin = [0] ++ [1]

coin2 :: [(Int, Int)]
coin2 = do
  x <- coin
  y <- coin
  return (x, y)
```

Bewaffnet mit diesen Ideen, modelliere einen fairen 6-seitigen Würfel. Berechne alle Möglichkeiten, wie man drei Würfel werfen kann, um die Augenzahl 11 zu erhalten.

Test 167 Implementiere das Prädikat `zip/3`, dass zwei Liste bekommt und eine Liste von Paaren zurückliefert – so wie du es aus Haskell kennst. Es soll

```
?- zip([1, 2], [3, 4, 5], [(1, 3), (2, 4)]).
```

gelten. Wie gewinnst du aus deiner Implementierung das Prädikat `unzip/3`, also die Umkehrfunktion `zip/3`, wenn diese auf Listen gleicher Länge eingeschränkt ist.

Test 168 Mithilfe von `append/3` lassen sehr viele andere Prädikate auf Listen definieren. Überlege dir, wie du folgende Prädikate unter dessen Nutzung implementieren kannst. Die Prädikate sollen sich wie deren entsprechenden Haskell-Funktionen verhalten, wenn dir der Name des Prädikats bekannt vorkommt.

- `head/2` und `tail/2`,
- `last/2`,
- `member/2` (entspricht `elem` in Haskell)
- `dups/2` soll alle Elemente in einer Liste finden, die genau zweimal vorkommen. Du darfst `not_member/2` als Hilfsfunktion verwenden.
- `sublist/2` soll erfüllbar sein, wenn eine Liste in einer anderen ohne Lücken enthalten ist.
- `subsequence/2` soll erfüllbar sein, wenn eine Liste in einer anderen mit möglichen Lücken enthalten ist.

Warum ist `member/2` hier zweistellig?

Test 169 Das Prädikat `sublist/2` soll genau dann erfüllbar sein, wenn eine gegebene Liste in einer anderen gegebenen Liste enthalten ist. Eine mögliche Implementierung sieht wie folgt aus:

```
is_prefix([], _).
is_prefix([X|Xs], [X|Ys]) :- is_prefix(Xs, Ys).

sublist(Xs, Ys) :- is_prefix(Xs, Ys).
sublist(Xs, [_|Ys]) :- sublist(Xs, Ys).
```

Was sind die Ergebnisse der Anfrage `?- sublist(Xs, [1, 2]).`?

Test 170 Warum ist `not_member(X, Xs) :- append(_, [Y|_], Xs), X \= Y.` keine korrekte Implementierung des Prädikates, das testet, ob ein Element nicht einer Liste enthalten ist?

Test 171 In diesem Selbsttest wollen wir die Peano-Arithmetik wiederholen. Implementiere folgende Prädikate:

- `peano/1` soll beweisbar sein, wenn der übergebene Term eine Peano-Zahl ist, also z.B. die Form `o`, `s(o)`, `s(s(o))`, ... hat.
- `add/3` soll die Addition auf Peano-Zahlen implementieren.
- Die Multiplikation kann wie folgt definiert werden:

  ```
  mult(o, _, o).
  mult(s(X), Y, Z) :- add(U, Y, Z), mult(X, Y, U).
  ```

  Eine alternative Implementierung könnte so aussehen:

  ```
  mult(o, _, o).
  mult(s(X), Y, Z) :- mult(X, Y, U), add(U, Y, Z).
  ```

  Welcher der beiden Implementierungen ist besser? Betrachte während deiner Überlegungen auch folgende Anfrage `?- mult(s(s(o)), Y, s(s(s(s(o))))).` – also sinngemäß die Anfrage $2 \cdot y = 4$.
- Implementiere weiter die Prädikate `lt/2`, `eq/2` für Peano-Zahlen, also die $<$-Relation und Gleichheit auf Peano-Zahlen.

Test 172 Das Sieb des Eratosthenes ist ein Algorithmus zur Bestimmung von Primzahlen. Dieses wollen wir nun in Prolog implementieren – mit Peano-Zahlen natürlich.

- Implementiere zuerst ein Prädikat `range/3`, das eine Liste berechnet, die alle Ganzzahlen in einem vorgegebenen Intervall enthält. Zum Beispiel soll `?- range(s(o), s(s(s(o))), [s(o), s(s(o))]).` beweisbar sein.
- Als Nächstes implementiere ein Prädikat `filter_by_prime/3`, das alle Elemente einer Liste entfernt, die durch eine gegebene Primzahl teilbar sind.
- Zuletzt benötigen wir noch `primes/2`, dass alle Primzahlen bis zu einer angegeben Zahl berechnen soll. Implementiere dieses mithilfe der bereits vorbereiteten Prädikate.
- Mit Test 173 und einem weiteren Prädikat `map_to_nat/2` kannst du die Liste der Primzahlen in eine lesbare Form bringen – du kannst auch ein allgemeines `map/3` mithilfe des Prädikats höherer Ordnung `call` implementieren.

Test 173 Implementiere ein Prädikat `to_nat/2`, das eine Peano-Zahl in eine natürliche Zahl konvertiert. Nutze dafür `is/2`. Wieso terminiert die Anfrage `?- to_nat(P, 3).` nicht?

Test 174 Ein Graph sei dargestellt als eine Liste von Kanten. Die Kanten seien wiederum als Tupel dargestellt.

Implementiere ein Prädikat `reachable/3`, das bestimmt, ob ein Knoten von einem anderen Knoten aus erreichbar ist. Du benötigst voraussichtlich ein weiteres vierstelliges Hilfsprädikat.

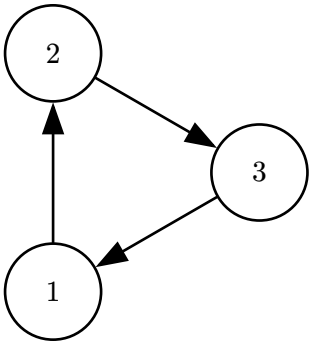
Dein Programm soll sich wie folgt verhalten:

```
?- reachable(1, 3, [(1, 2), (2, 3), (3, 1)]).
true ;
```

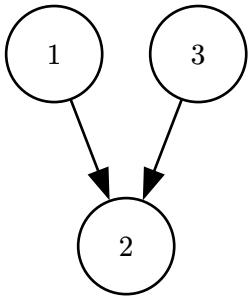
```
false.

?- reachable(1, 3, [(1, 2), (3, 2)]).
false.
```

Hier sind die Eingabe-Graphen nochmal visualisiert.



Eingabe-Graph der ersten Anfrage mit unendlich vielen gerichteten Pfaden von 1 bis 3



Eingabe-Graph der zweiten Anfrage mit keinem gerichteten Pfad von 1 nach 3

Test 175 Wie unterscheiden sich die Gleichheit `==/2` in Prolog und die Gleichheit in Haskell?

Test 176 Welche Konzepte, die Haskell verwendet, stecken hinter den Anfragen

- `?- just((X, Y)) = just((1, 2))` und
- `?- to(X, to(X, list(X))) = to(int, to(int, list(int)))`?

Test 177 Eine Dur-Tonleiter hat folgendes Muster relativ zum vorherigen Ton in Halbtonschritten:

$$+2,\ +2,\ +1,\ +2,\ +2,\ +2,\ +1$$

Zum Beispiel erhalten wir mit dem Grundton C, so die C-Dur Tonleiter

$$\text{C - D - E - F - G - A - B - C.}$$

D ist zwei Halbtöne von C entfernt, E ist zwei Halbtöne von D entfernt, F ist einen Halbton von E entfernt und so weiter.

Schreibe ein Prädikat `all_major/2`, dass alle diatonischen Dur-Akkorde (wie in Test 163 definiert) einer Dur-Tonleiter mit gegebenen Grundton berechnet. Diatonisch bedeutet, dass der Akkord nur aus Tönen der gegebenen Tonleiter bestehen darf. Für die C-Dur Tonleiter sind die gesuchten Akkorde C-Dur, F-Dur und G-Dur.

Test 178 Entwickle ein Prädikat `nth/3`, dass das n-te Element einer Liste zurückgibt. Zum Beispiel soll folgende Anfrage beweisbar sein.

```
?- nth([3, 1, 4, 1, 5], s(s(o)), X).
X = 4.
```

Test 179 Für die nächste Ausgabe des Mittelerde-Kuriers benötigst du noch ein Titelbild für den Artikel „Die Gefährten ziehen zum Schicksalsberg“. Dafür möchtest du die Gefährten der Größe nach aufstellen. Merry und Pippin sind sich nicht einig darüber, wer der größere der beiden ist. Du entscheidest dich, beide Varianten zu fotografieren.

Die Größen-Relation ist mithilfe des folgenden Prädikats festgehalten.

```
before(X, Y, Xs) :- append(_, [X|R], Xs), append(_, [Y|_], R).
```

```
smaller(X, Y) :- before(X, Y, [frodo, merry, pippin, sam]).
smaller(pippin, merry).
```

- Implementiere ein Prädikat `insert/4`, dass ein Element bzgl. einer Ordnung in eine gegebene Liste einfügt.

  ```
  ?- insert(smaller, merry, [pippin], Fs).
  Fs = [merry, pippin] ;
  Fs = [pippin, merry].
  ```

  Falls dir Prädikate höherer Ordnung Schwierigkeiten bereiten, kannst du auch zuerst versuchen, eine Implementierung mit einer festen $<$-Relation angeben.
- Implementiere mithilfe des `insert`-Prädikats Sortieren durch Einfügen. Das Prädikat sollte dann beide Sortierungen ausgeben.

  ```
  ?- isort(smaller, [sam, pippin, frodo, merry], Fs).
  Fs = [frodo, merry, pippin, sam] ;
  Fs = [frodo, pippin, merry, sam] ;
  ```

Test 180 Aus welchen Teilen besteht das generate-and-test Schema?

Test 181 Wie hängen musterorientierte Prädikate und induktiv definierte Funktionen miteinander zusammen?

Challenge 20 Das Erfüllbarkeitsproblem der Aussagenlogik fragt, ob es für eine gegebene aussagenlogische Formel eine Belegung der Variablen mit wahr oder falsch gibt, sodass die Formel insgesamt wahr ist.

Zur Erinnerung, die Syntax aussagenlogischer Formeln ist wie folgt beschrieben:
- $\top$ (true), $\bot$ (false) und Variablen sind aussagenlogische Formeln.
- Seien F, G aussagenlogische Formeln, dann sind
  - $\neg F$ (Negation),
  - $F \wedge G$ (Konjunktion) und
  - $F \vee G$ (Disjunktion)

  aussagenlogische Formeln.

Die Semantik der atomaren Formeln und der Junktoren wird durch die entsprechenden booleschen Funktionen und eine Belegung der Variablen definiert.

- Implementiere zuerst `bool/1`, das den Wertebereich festlegen soll, und die booleschen Funktionen `and/3`, `or/3`, `neg/2`.
- Implementiere ein Prädikat `get_vars/2`, das alle Bezeichner von (freien) Variablen einer gegebenen Formel zurückgibt. Eine Variable soll in einer Formel als Term `var(X)` gekennzeichnet werden. `X` ist dann der Bezeichner der Variable. Stelle dabei sicher, dass die Liste der Bezeichner keine Duplikate enthält – dafür kannst du z.B. `list_to_set/2` verwenden.
- Implementiere als Nächstes ein Prädikat `assignment/2`, das alle Belegungen generiert. Eine Zuweisung soll als Liste von Tupeln dargestellt werden (vgl. Beispielanfragen).
- Implementiere ein Prädikat `eval/3`, das beweisbar ist, wenn die gegebene Formel erfüllbar unter der gegegeben Belegung ist. Hier sind ein paar Beispielanfragen:

  ```
  ?- eval(and(true, var(x)), [(x, true)], false).
  false.

  ?- eval(and(true, var(x)), [(x, B)], false).
  B = false.
  ```

- Implementiere ein Prädikat `sat/2`, das alle belegten Formeln berechnet. Hier sind ein paar weitere Beispielanfragen:

```
?- sat(or(var(x), neg(var(x))), A).
A = [(x, true)] ;
A = [(x, false)] ;
false.

?- sat(and(var(x), or(neg(var(y)), neg(var(z)))), A).
A = [(x, true), (y, true), (z, false)] ;
A = [(x, true), (y, false), (z, true)] ;
A = [(x, true), (y, false), (z, false)] ;
false.

?- sat(and(var(x), neg(var(x))), A).
false.
```

Test 182 In diesem Test ergänzen wir den SAT-Löser aus Challenge 20 um die Implikation und Äquivalenz. Anstatt das `eval`-Prädikat entsprechend zu erweitern, wollen wir diese Terme in andere Terme übersetzen, die wir bereits auswerten können. Das können wir mit folgenden Regeln erreichen:

$$(A \Rightarrow B) \quad \Leftrightarrow \quad (\neg A \vee B) \qquad \text{und} \qquad (A \Leftrightarrow B) \quad \Leftrightarrow \quad (A \Rightarrow B \wedge B \Rightarrow A)$$

Implementiere dafür ein Prädikat `desugar/2`. Als Konstruktoren kannst du z.B. `impl/2` und `iff/2` verwenden.

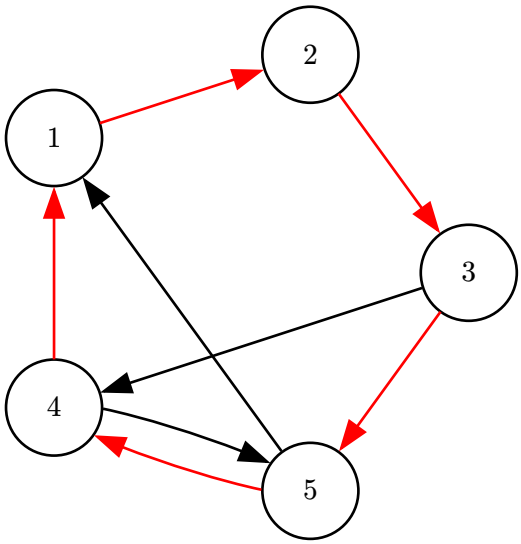
Test 183 Wir machen uns auf in die Kombinatorik und wollen ein paar nützliche Prädikate definieren, die uns helfen Suchräume zu durchlaufen.

- Implementiere ein Prädikat `varia_rep/3`, das genau dann beweisbar ist, wenn eine über gegebene Liste eine Variation einer anderen ist und aus k Elementen besteht – es soll z.B. `?- varia_rep([0, 1], 4, [1, 0, 0, 1]).` beweisbar sein.
- Implementiere ein Prädikat `perms/2`, dass genau dann beweisbar ist, wenn zwei übergebene Listen Permutationen voneinander sind.

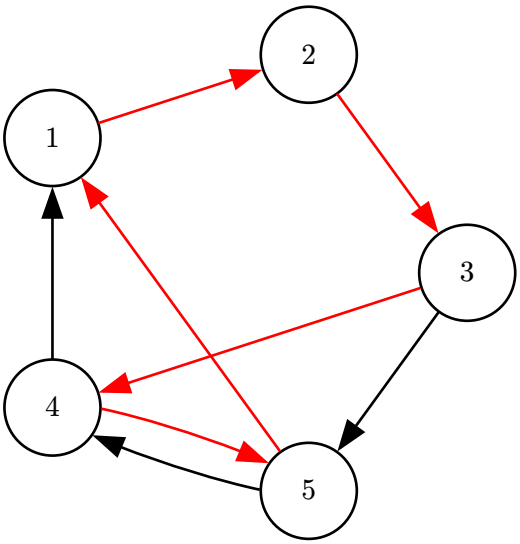
Challenge 21 Oft eignet sich in Prolog gut, um Algorithmen für Entscheidungsprobleme, die in NP liegen, zu implementieren. Als Methode dafür hast du generate-and-test kennengelernt – wir generieren mögliche (also korrekte oder falsche) Lösungen und entscheiden dann, ob sie korrekt sind.

Ein Hamiltonkreis ist ein Kreis in einem Graph, der jeden Knoten genau einmal enthält. Wir wollen diese als Listen darstellen. Durch Rotation dieser Liste erhalten wir äquivalente Hamiltonkreise. Diese wollen wir ignorieren und wählen stattdessen einen kanonischen Repräsentanten, indem wir uns auf die Liste beschränken, die mit dem kleinsten Knoten beginnt.

Im folgenden Graph gibt es zwei Hamiltonkreise.



Hamiltonkreis 1: $(1, 2, 3, 5, 4)$



Hamiltonkreis 2: $(1, 2, 3, 4, 5)$

Um Hamiltonkreise zu bestimmen, arbeiten wir uns von einem naiven Pfad-Generator zu einem, der versucht nach und nach den Suchraum durch vorhandene Informationen zu verkleinern (auch pruning genannt). Danach verifizieren wir, ob ein gegebener Pfad ein Hamiltonkreis ist. Graphen kannst du z.B. als Liste von Kanten darstellen.

```
graph([(1, 2), (2, 3), (3, 4), (3, 5), (4, 5), (4, 1), (5, 1), (5, 4)]).
```

Es lohnt sich, Test 183 vor dieser Challenge gemacht zu haben, wenn du mit den naiveren Pfad-Generatoren starten möchtest.

- Implementiere zuerst einen (polynomiellen) Verifizierer, also ein Prädikat `is_hamilton/2`, das bestimmt, ob ein gegebener Pfad in einem gegebenen Graphen einem Hamiltonkreis entspricht.
- Implementiere ein Prädikat `path/2` und verbessere sukzessiv mit den folgenden Ideen:
  - Prüfe jede beliebige Knotenfolge bestehend aus $|V|$ Knoten. Das heißt, der Suchraum entspricht zu Beginn $V^{|V|}$ für einen Graphen (V, E).
  - Jede zwei Knoten dieser Folge müssen paarweise unterschiedlich sein.
  - Jede zwei aufeinanderfolgenden Knoten entsprechen einer Kante in dem Eingabegraphen. Die Knotenfolge ist also ein Pfad.
- Implementiere zuletzt das Prädikat `hamilton/2`, das Hamiltonpfade bzw. -kreise berechnet.

Beobachte experimentell, wie sich die Laufzeit durch die einzelnen Pruning-Schritte verändert.

Im Folgenden stehen Großbuchstaben für Variablen und Kleinbuchstaben für atomare Ausdrücke – wenn nicht anders in Test oder Challenge eingeführt.

Test 184 Was besagt die Abtrennungsregel (modus ponens)?

Test 185 Was ist das (einfache) Resolutionsprinzip?

Test 186 Wann ist eine Anfrage mithilfe des Resolutionsprinzips beweisbar?

Test 187 Mithilfe welcher Methode stellen wir fest, ob ein Literal zu einer linken Regelseite passt?

Test 188 Wie sind Terme definiert?

Test 189 Welche der folgenden Terme sind syntaktisch korrekt?

- X

- a
- XX
- $f(X, a)$
- $f(f(X))$
- $g(f(X)(Y))$

Test 190 Mithilfe welcher Methode ersetzen wir Variablen in Termen?

Test 191 Wie ist die Substitution auf Termen definiert? Was wird insbesondere durch eine Substitution verändert und was nicht?

Test 192 Falls du Challenge 20 gemeistert hast – an welcher Stelle deines Programms führst du eine Substitution durch?

Test 193 Sei $\sigma = \{X \mapsto 1, Y \mapsto 2\}$ eine Substitution. Welche Anwendungen oder Aussagen sind korrekt?
- $\sigma(\text{add}(X, Y)) = \text{add}(1, 2)$
- $\sigma(\text{eq}(X, X)) = \text{eq}(1, X)$
- $\sigma(f(g(X, Y), Z))$ ist nicht definiert.

Test 194 Welche der folgenden Substitutionen sind wohldefiniert?
- $\sigma = \{X \mapsto 1\}$
- $\sigma = \{X \mapsto X\}$
- $\sigma = \{f(X) \mapsto f(Y)\}$
- $\sigma = \{X \mapsto Y, Y \mapsto X\}$
- $\sigma = \{[X|Y] \mapsto [1|[]]\}$

Test 195 Sind $[X]$ und $[1, 2]$ unifizierbar?

Test 196 Wende die Substitution $\sigma = \{X \mapsto 1, Y \mapsto f(X)\}$ auf den Term $g(X, h(Y))$ an, ohne einen Zwischenschritt auszulassen.

Test 197 Was ist ein Unifikator? Was ist ein allgemeinster Unifikator?

Test 198 Was sind die Ergebnisse der folgenden Kompositionen von Substitutionen?
- $\{Y \mapsto X\} \circ \{Z \mapsto 1\}$
- $\{Y \mapsto X\} \circ \{Z \mapsto Y\}$
- $\{Z \mapsto Y\} \circ \{Y \mapsto X\}$

Test 199 Beim Durchführen des Unifikationsalgorithmus treten in zwei verschiedenen Iterationen i, j mit $i < j$ die Substitutionen

$$\sigma_i = \{X \mapsto f(Y)\} \text{ und } \sigma_j = \{X \mapsto a\}$$

auf. Ist ein solcher Verlauf korrekt? Begründe deine Antwort. Dabei kann angenommen werden, dass X nicht in Y vorkommt.

Test 200 Warum gilt $\{Y \mapsto X\} \circ \{Z \mapsto Y\} = \{Z \mapsto Y\} \circ \{Y \mapsto X\}$ nicht?

Test 201 Warum ist $\{Y \mapsto X\} \circ \{Y \mapsto 2\} = \{Y \mapsto 2\}$?

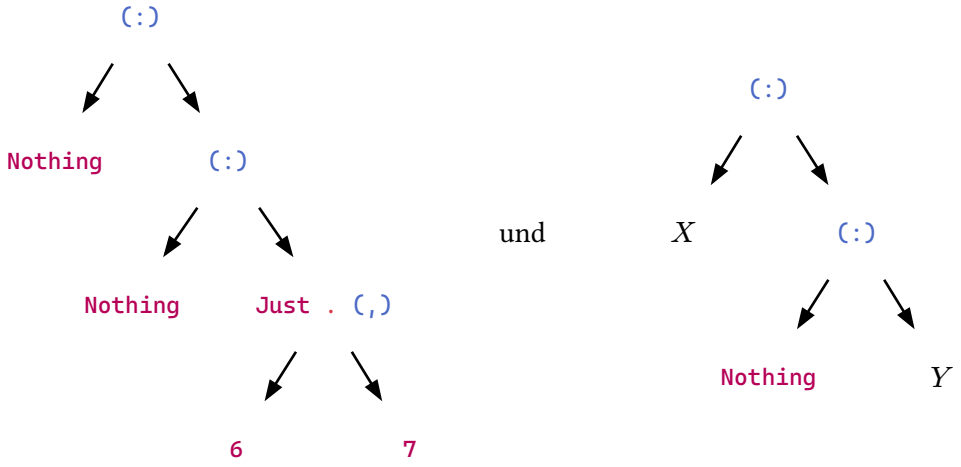
Test 202 Warum ist die Komposition von Substitutionen im Allgemeinen nicht die Vereinigung der jeweiligen Mengendarstellungen?

Test 203 Wann existiert ein allgemeinster Unifikator?

Test 204 Was ist die Unstimmigkeitsmenge zweier Terme, was gibt sie an und wie ist sie definiert?

Test 205 Welche Unstimmigkeitsmengen sind korrekt berechnet?
- $\mathrm{ds}(f(X), f(1)) = \{X, 1\}$
- $\mathrm{ds}(1, 2) = \{1, 2\}$
- $\mathrm{ds}(g(1, 2), h(1, 2)) = \emptyset$
- $\mathrm{ds}(X, Y) = \emptyset$
- $\mathrm{ds}(f(X, Y), f(1, 2)) = \{Y, 2\}$

Test 206 Finde Unifikatoren für die folgende Terme:
- $(X + 1) \cdot Y + Z$ und $((3 + Z) + 1) \cdot Z + 3$, und
-



Test 207 Unter welchen Umständen terminiert der Unifikationsalgorithmus?

Test 208 Aus welcher Eigenschaft des Unifikationsalgorithmus folgt, dass ein berechneter Unifikator ein allgemeinster Unifikator ist?

Test 209 Seien t_1, t_2 Terme. Wenn $\mathrm{ds}(\sigma(t_1), \sigma(t_2)) = \emptyset$ gilt, was können wir über σ folgern?

Test 210 Welche Arten von Fehlschlägen können während des Unifikationsalgorithmus auftreten? Unter welchen Umständen treten diese auf?

Test 211 In welches Problem laufen wir, wenn wir mit einer Substitution, die sich aus $\mathrm{ds}(t_1, t_2) = \{X, f(X)\}$ ergibt, naiv weiter rechnen würden, wobei t_1 und t_2 Terme sind?

Test 212 Was bedeutet es, wenn der Vorkommenstest (occurs check) positiv ist?

Test 213 Gebe ein Beispiel für eine Eingabe an, für das der Unifikationsalgorithmus exponentielle Laufzeit bzgl. der Größe der Eingabeterme hat.

Die Größe eines Terms $|\cdot|$ wir z.B. wie folgt berechnen:
- $|X| = 1$, falls X Variable ist,
- $|a| = 1$, falls a Konstante ist und
- $|f(t_1, ..., t_n)| = 1 + \sum_{i=1}^{n} |t_i|$ für Terme $t_1, ..., t_n$ und n-stelligen Funktor f.

Test 214 Wieso kommt der Fall der exponentiellen Laufzeit in der Größe der Eingabeterme überhaupt zustande?

Test 215 Aus welchen Komponenten setzt sich das allgemeine Resolutionsprinzip zusammen? Wie wird es auch genannt?

Test 216 Was legt die Selektionsfunktion der SLD-Resolution fest? Welche verwendet Prolog?

Test 217 Gegeben sei die Anfrage ?- $A_1, ..., A_m$. Du stellst fest, dass A_1 mit der linken Seite der Regel L :- $L_1, ..., L_n$ unifizierbar ist. Welche Anfrage ist in einem SLD-Resolutionsschritt daraus ableitbar.

Test 218 Wieso benennen wir Variablen einer Regel um, bevor wir eine Unifikation als Teil eines SLD-Resolutionsschritt durchführen?

Test 219 Wodurch ergibt sich die Struktur eines SLD-Baums?

Test 220 Welche Auswertungsstrategie findet immer eine Lösung, falls eine existiert?

Test 221 Warum wird die Tiefensuche als Auswertungsstrategie der Breitensuche bevorzugt?

Test 222 Wie ergibt sich die Reihenfolge der Kindknoten eines Knoten in einem SLD-Baum?

Test 223 Welche Rolle spielt Backtracking in Prolog?

Test 224 Wie werden Variablen in Prolog gebunden?

Test 225 Wieso wird empfohlen, dass Klauseln für Spezialfälle vor allgemeineren Klauseln stehen sollten?

Test 226 Gegeben sei das Prolog-Programm.

```
fruit(apple).
fruit(banana).
fruit(cranberry).

salad(F1, F2, F3) :- fruit(F1), fruit(F2), fruit(F3).
```

Was sind die ersten vier Lösungen, die Prolog berechnet? Warum verändern sich die Belegungen für `F2` und `F3` zuerst?

Test 227 Gegeben sei das Prolog-Programm:

```
nth(0, [X|_], X).
nth(N, [_|Xs], X) :- N > 0, M is N - 1, nth(M, Xs, X).
```

Wir beginnen, eine Lösungen zu berechnen, wie Prolog sie berechnen würde, für die Anfrage `?- nth(1, [3, 1, 4, 1]).`.

```
?- nth(1, [3, 1, 4, 1]).
|- (2. Regel)
?- 1 > 0, M is 1 - 1, nth(M, [1, 4, 1], X).
|- (2. Regel)
?- 1 > 0, M is 1 - 1, M > 0, M is M - 1, nth(M, [4, 1], X).
|-
...
```

Was ist hier schief gelaufen?

Test 228 Begründe warum die folgende Aussage korrekt oder falsch ist: Beim Berechnen der Unstimmigkeitsmenge kann folgendes Ergebnis herauskommen:

$$\{g(X, Y), g(Y, X)\}.$$

Test 229 Gegeben sei folgendes Prolog-Programm.

```
append([],    L, L     ).
append([E|R], L, [E|RL]) :- append(R, L, RL).
```

Es wird die Anfrage `?- append(X, Y, [1, 2]).` gestellt. Beim Anwenden der zweiten Regel wurde die Substitution

$$\sigma_1 = \{X \mapsto [E_1|R_1], Y \mapsto L_1, E_1 \mapsto 1, RL_1 \mapsto [2]\}$$

berechnet. Ist diese Substitution ein Unifikator für `append(X, Y, [1, 2])` und `append([E1|R1], L1, [E1|RL1])`, der aus dem Unifikationsalgorithmus entstanden ist?

Test 230 Gegeben sei folgendes Prolog-Programm.

```
f(X, 2).
```

Welcher Schritt der SLD-Resolution ist hier essentiell, um eine korrekte Lösung zu erhalten. Betrachte dafür die folgende Anfrage `f(1, X).`

Zu den am häufigsten gemachten Fehlern beim Angeben eines SLD-Baums ist die fehlerhafte Anwendung des Unifikationsalgorithmus. Beim intuitiven Anwenden des Algorithmus suchen wir oftmals nur die Stellen in den Ausgangstermen, an denen Variablen auf Terme stoßen und vergessen, zuvor freie und bereits gebundene Variablen zu ersetzen. Als Unifikator für z.B. `f(1, Y)` und `f(X, [X|R])` wird oft

$$\sigma = \{X \mapsto 1, Y \mapsto [X|R]\}$$

angegeben. Das ist falsch, da

$$\sigma(f(1, Y)) = f(1, [X|R]) \neq f(1, [1|R]) = \sigma(f(X, [X|R]))$$

Auf der linken Seite könnten wir den Unifikator zwar erneut anwenden, um den korrekten Term zu erhalten, damit halten wir uns allerdings nicht an die Definition der Unifizierbarkeit.

Wenn wir den Unifikationsalgorithmus auf die beiden Terme formal anwenden, sehen wir in der ersten Iterationen, dass der Teilterm $[X|R]$ nicht mehr in $\sigma_1(t_2) = f(1, [1|R])$ vorkommt.

k	σ_k	$\sigma_k(t_1)$	$\sigma_k(t_2)$	$\mathrm{ds}(\sigma_k(t_1), \sigma_k(t_2))$
0	$\emptyset$	$f(1, Y)$	$f(X, [X\|R])$	$\{X, 1\}$
1	$\{X \mapsto 1\}$	$f(1, Y)$	$f(1, [1\|R])$	$\{Y, [1\|R]\}$
2	$\{X \mapsto 1, Y \mapsto [1\|R]\}$	$f(1, [1\|R])$	$f(1, [1\|R])$	$\emptyset$

Wenn wir die Anwendung der Substitution σ_1 auf den Teilterm $[X|R]$ von t_2 verspäten, sehen wir, wo der obige Fehler liegt.

$$\begin{aligned}\sigma_2 \circ \sigma_1 &= \{Y \mapsto \sigma_1([X|R])\} \circ \{X \mapsto 1\} \\ &= \{Y \mapsto [1|R], X \mapsto 1\}.\end{aligned}$$

Wie können wir daran denken? Wenn eine Variable X in der k. Iteration gebunden wird, dann kann X weder in $\sigma_k(t_1)$ noch in $\sigma_k(t_2)$ vorkommen. Für dieses Beispiel bedeutet das, wenn wir kurz davor sind, $Y \mapsto [X|R]$ hinzuschreiben, sollten wir uns vergewissern, ob bereits gebundene Variablen auf der rechten Seite vorkommen und entsprechend ersetzen. In einem anderen Fall können wir Variablen haben, die bereits an Terme gebunden sind aber noch freie Vorkommen von Variablen haben – betrachte z.B. $f(X, 1)$ und $f([Y|R], Y)$ mit $\{Y \mapsto 1\} \circ \{X \mapsto [Y|R]\}$. Hier müssen andersherum schauen, ob die neue Belegung alte Belegungen verändert. Das geht aus der linken Menge der Definition der Komposition hervor.

$$\varphi \circ \psi = \{v \mapsto \varphi(t) \mid v \mapsto t \in \psi, \varphi(t) \neq v\} \cup \{v \mapsto t \mid v \mapsto t \in \varphi, v \notin D(\psi)\}.$$

Zu beachten ist, dass

$$\{Y \mapsto [X|R]\} \circ \{X \mapsto 1\} = \{Y \mapsto [X|R], X \mapsto 1\} \neq \{Y \mapsto [1|R], X \mapsto 1\}.$$

Die linken Substitutionen ist keine korrekten Zwischenschritte.

Test 231 Die Komposition von Substitutionen ist definiert durch

$$\varphi \circ \psi = \{v \mapsto \varphi(t) \mid v \mapsto t \in \psi, \varphi(t) \neq v\} \cup \{v \mapsto t \mid v \mapsto t \in \varphi, v \notin D(\psi)\}.$$

Wieso spielen $\varphi(t) \neq v$ und $v \notin D(\psi)$ keine Rolle, wenn wir den akribisch Unifikationsalgorithmus anwenden?

Test 232 Was verbirgt sich hinter dem Begriff „Negation als Fehlschlag“?

Test 233 Wann ist `\+ p` beweisbar?

Test 234 Wieso stimmt die Negation als Fehlschlag nicht mit der prädikatenlogischen Negation überein?

Test 235 Warum sollte `p` keine freien Variablen enthalten, wenn wir `\+ p` beweisen wollen? Wieso ergibt sich daraus die Empfehlung, dass Negationen soweit wie möglich rechts in einer Regel stehen sollten?

Test 236 Wie sind die Variablen `X, Y` in der folgenden Regel quantifiziert?

```
p(X) :- q(X, Y).
```

Test 237 Nutze $\forall x : p \Leftrightarrow \neg(\exists x : \neg p)$, um ein Prädikat `forall/2` zu definieren.

Mithilfe dieses Prädikates soll es möglich sein, folgende Anfrage auszudrücken.

```
?- forall(member(X, [1, 2, 3]), X > 0).
```

Die Anfrage bedeutet, für alle $x \in \{1, 2, 3\}$ gilt $x > 0$.

Wie können wir mit `forall` die Definition von `not_member` aus Test 170 reparieren?

Test 238 Wofür verwenden wir den Cut-Operator konzeptionell?

Test 239 Wann ist `p` in `p :- q, !, r.` beweisbar? Was ist insbesondere der Fall, wenn `q` beweisbar ist?

Test 240 Wir ergibt sich aus der Semantik des Cut-Operators ein Fallunterscheidung-Konstrukt?

Test 241 Wie können wir mithilfe des Cut-Operators die Negation als Fehlschlag implementieren?

Test 242 Wie wirkt sich der Cut-Operator auf die Struktur eines SLD-Baums aus?

Test 243 Wie können wir mithilfe des Cut-Operators im folgenden Prädikat Berechnungen sparen?

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- X < Y.
```

Test 244 Wofür wird das Prädikat `is/2` verwendet?

Test 245 Welche Anfragen sind valide?

- `X is 42 - 3`
- `X is Y + 1`
- `Y = 1, X is Y + 1`
- `32 + 10 is X`

- `42 is 40 + Y`

Test 246 Warum können wir `=/2` nicht für Arithmetik verwenden bzw. wieso gilt `42 is 40 + 2` nicht?

Test 247 Welche weiteren Prädikate kennst du neben `is/2` die auch Terme ausrechnen?

Test 248 Ist `X =:= 4 + 7.` eine valide Anfrage in Prolog?

Test 249 Implementiere ein Prädikat `all_trees/2`, das jeden möglichen blätterbeschriften Binärbaum erzeugt, deren Blätter von links nach rechts die Eingabeliste lesen.

```
?- all_trees([1, 2, 3], Ts).
Ts = [
  branch(leaf(1), branch(leaf(2), leaf(3))),
  branch(branch(leaf(1), leaf(2)), leaf(3))
].
```

- Implementiere zuerst eine Variante, in der du auf keine besonderen Hilfsprädikate zurückgreifst.
- Gegeben sei folgende fehlerhafe Implementierng des Prädikats.

  ```
  all_trees([], []).
  all_trees([X], [leaf(X)]) :- !.
  all_trees(Xs, Ts) :-
    member(branch(Lt, Rt), Ts),
    append([L|Ls], [R|Rs], Xs),
    all_trees([L|Ls], Lts),
    member(Lt, Lts),
    all_trees([R|Rs], Rts),
    member(Rt, Rts).
  ```

  Auf dem ersten Blick scheint eine elegante Idee hinter dieser Definition zu stecken. Wieso funktioniert diese Idee allerdings nicht?
- Korrigiere die fehlerhafte Definition unter Verwendung von `findall/2` und erhalte dabei grundsätzliche Idee.
- Staune darüber, wie vergleichweise müheselig das Implementieren der ersten Variante ohne Hilfsprädikate war, und sei dankbar für `findall/3`.

Hier sind ein paar Logik-Puzzle, die zu deiner Belustigung dienen. Der Lerneffekt ist voraussichtlich sehr gering.

Challenge 22 Das Zebra-Rätsel ist ein Logikpuzzle.

So wird es auf Wikipedia wiedergegeben:

1. Es gibt fünf Häuser.
2. Der Engländer wohnt im roten Haus.
3. Der Spanier hat einen Hund.
4. Kaffee wird im grünen Haus getrunken.
5. Der Ukrainer trinkt Tee.
6. Das grüne Haus ist direkt rechts vom weißen Haus.
7. Der Raucher von Old-Gold-Zigaretten hält Schnecken als Haustiere.
8. Die Zigaretten der Marke Kools werden im gelben Haus geraucht.
9. Milch wird im mittleren Haus getrunken.
10. Der Norweger wohnt im ersten Haus.
11. Der Mann, der Chesterfield raucht, wohnt neben dem Mann mit dem Fuchs.

12. Die Marke Kools wird geraucht im Haus neben dem Haus mit dem Pferd.
13. Der Lucky-Strike-Raucher trinkt am liebsten Orangensaft.
14. Der Japaner raucht Zigaretten der Marke Parliaments.
15. Der Norweger wohnt neben dem blauen Haus.

Wer trinkt Wasser? Wem gehört das Zebra?

Löse dieses Puzzle mithilfe von Prolog und übersetze dabei so wenig logische Schlüsse, die du selbst in deinem Kopf machst, in deinem Programm.

Challenge 23 Ein Logikpuzzle, das einst viral ging, ist bekannt als „Cheryl's Geburtstag". Es ist das erste Puzzle aus dem paper Cheryl's Birthday. Löse dieses Puzzle mithilfe von Prolog und übersetze dabei so wenig logische Schlüsse, die du selbst in deinem Kopf machst, in deinem Programm.

Hinweise zu Tests und Challenges

Hinweis zu Challenge 1

- Zur Darstellung der Multimengen eignen sich sortierte Listen gut.
- 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 5 Nach n `enqueue`- Operationen in eine leere queue sieht diese wie folgt aus

```
Q [] [xn, ..., x1].
```

Wenn als Nächstes `dequeue`-Operationen naiv ausgeführt werden, müssten wir die `enqueue`-Liste jedes Mal vollständig durchlaufen. Wenn wir die `dequeue`-Liste entsprechend verändern, können wir mindestens die nächsten n `dequeue`-Operationen effizient durchführen.

Hinweis zu Challenge 10 Wenn du Anlaufschwierigkeiten hast, helfen dir möglicherweise diese ersten Implementierungen weiter.

```
instance Num a => Num (D a) where
  D x1 d1 + D x2 d2 = D (x1 + x2) (d1 + d2)  -- Summenregel
  -- ...

instance Fractional a => Fractional (D a) where
  -- ...

instance Floating a => Floating (D a) where
  exp (D x d) = D (exp x) (d * exp x)  -- Kettenregel und exp'(x) = exp(x)
  -- ...
```

Hinweis zu Challenge 15 Die Rekursionsvorschrift ist gegeben durch

$$\text{pathsum}(i,j) = \begin{cases} G_{0,0} & \text{falls } (i,j) = (0,0) \\ G_{i,0} + \text{pathsum}(i-1,0) & \text{falls } j = 0 \\ G_{0,j} + \text{pathsum}(0,j-1) & \text{falls } i = 0 \\ G_{i,j} + \min(\text{pathsum}(i-1,j), \text{pathsum}(i,j-1)) & \text{sonst} \end{cases}$$

für $i \in \{0,\ldots,m-1\}, j \in \{0,\ldots,n-1\}$. Die minimale Pfadsumme ist dann $\text{pathsum}(m-1,n-1)$.

Hinweis zu Test 139 Die Implementierung dieser Monade ist sehr ähnlich zu der Implementierung der `Reader`-Monade, wie sie in Test 138 angegeben ist.

Hinweis zu Challenge 16 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 17 Falls du es nicht geschafft hast, in Test 152 `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
- Prolog Cheatsheet

Appendix

Bisher findest du hier im Anhang des Dokuments Bemerkungen zu verschiedenen Tests oder Challenges. Du kannst alles ignorieren, was hier steht. Die Bemerkungen sind aus Neugier entstanden und sollen letztendlich in dir höchstens ein „Interessant! 🤔“ auslösen. Danach darfst du direkt wieder vergessen, was hier steht.

Bemerkung 1 In Challenge 10 wurden Funktionen `d1`, `d2` und `d3` definiert. Diese ver- und entschachteln `D`-Werte unterschiedlich tief. Mit verschiedenen Haskell-Spracherweiterungen können wir eine allgemeinere Funktion angeben. Mit `derive @3 cos (2.0 :: Double)` können wir jetzt z.B. die dritte Ableitung des Cosinus an der Stelle 2 berechnen.

```
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE AllowAmbiguousTypes #-}
{-# LANGUAGE FunctionalDependencies #-}

import Data.Kind
import GHC.TypeLits

data D a = D a a

-- Hier könnten deine Num-, Fractional- und Floating-Typklasseninstanzen
-- stehen!

class Derivable (n :: Nat) b where
  type Derivative n b
  derive :: (Derivative n b → Derivative n b) → b → b

instance (Wrap (NatToPeano n) b, Unwrap (DK (NatToPeano n) b) b) ⇒ Derivable n b
where
  type Derivative n b = DK (NatToPeano n) b
  derive f x = unwrap (f (wrap @(NatToPeano n) x))

data Peano = Z | S Peano

type family NatToPeano (n :: Nat) where
  NatToPeano 0 = Z
  NatToPeano k = S (NatToPeano (k - 1))

type family DK p a where
  DK Z     a = a
  DK (S k) a = D (DK k a)

class Wrap n t where
  wrap :: t → DK n t

instance Wrap Z t where
  wrap x = x

instance (Wrap n t, Num (DK n t)) ⇒ Wrap (S n) t where
  wrap x = D (wrap @n x) 1

class Unwrap a b | a → b where
  unwrap :: a → b

instance (Num a) ⇒ Unwrap a a where
```

```haskell
  unwrap x = x

instance (Unwrap a b, Num a) => Unwrap (D a) b where
  unwrap (D _ d) = unwrap d
```

Bemerkung 2 In Test 74 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).

```java
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
  }
}
```