LISTEN

```haskell
-- fromIntegral dient dazu, eine Ganzzahl in eine Gleitkommazahl zu konvertieren.
-- sum berechnet die Summe der Elemente in einer Liste.
-- length gibt die Anzahl der Elemente in einer Liste zurück.
meanListe :: [Int] -> Float
meanListe mean = fromIntegral (sum mean) / fromIntegral (length mean)

list :: [(Integer, Integer)]
list = [(i,100-1)| i <-[0..99]]

list2 :: [(Integer, Integer, Integer)]
list2 = [(x,y,z) |
  x <- [1..50],
  y <- [x..(100-x)],
  z<- [100 - x - y],
  z >= y
  ]

-- prims ist eine unendliche Liste von Primzahlen, die durch die Funktion "sieve" erzeugt wird.
-- "2.." erzeugt eine unendliche Liste von Zahlen, die bei 2 beginnt. Ohne obere Grenze.
-- "sieve" ist eine Funktion, die eine Liste von Primzahlen erzeugt.
-- "where" wird verwendet, um lokale Funktionen zu definieren.
-- "sieve [] = []" ist der Basisfall, der eine leere Liste zurückgibt.
-- "sieve (p:xs)" ist der rekursive Fall, der die Primzahl p aus der Liste entfernt und die Liste
rekursiv filtert.
-- die ()-Syntax wird verwendet, um eine Liste zu dekonstruieren.
-- p:xs dekonstruiert die Liste in das erste Element p und den Rest der Liste xs.
-- "p : sieve [x | x <- xs, x `mod` p /= 0]" fügt die Primzahl p zur Liste hinzu und filtert alle
Vielfachen von p aus der Liste.
-- p : sieve [] entfernt alle Vielfachen von p aus der Liste.
-- "x `mod` p /= 0" überprüft, ob x ein Vielfaches von p ist.
-- x <- xs durchläuft alle Elemente in der Liste xs.
prims :: Integral a => [a]
prims = sieve [2..]
  where
    sieve :: Integral a => [a] -> [a]
    sieve [] = []
    sieve (p:xs) = p : sieve [x | x <- xs, x `mod` p /= 0]

-- einfachere Lösung
prims_2 :: Integral a => [a]
prims_2 =
  [ x | x <- [2 ..],
    null [ y | y <- [2 .. x `div` 2], x /= y, x `mod` y == 0]]

-- "prim" ist eine Funktion, die die x-te Primzahl zurückgibt.
-- "!!" ist ein Operator, der das Element an einem bestimmten Index in einer Liste zurückgibt.
-- "x - 1" wird verwendet, um den Index in der Liste zu berechnen, da die Indizes in Haskell bei 0
beginnen.
prim :: Integral a => Int -> a
prim x = prims !! (x - 1)

-- einfachere Lösung
prim_2 :: Integral a => Int -> a
prim_2 = (prims !!) . flip (-) 1

--Aufgabe 1: Listenverkettung
--Konzept: Verkettung von Listen (++)
--Aufgabe: Schreiben Sie eine Funktion concatLists :: [a] -> [a] -> [a], die zwei Listen verkettet.
--Erklärung: Die ++-Operation verbindet zwei Listen zu einer einzigen Liste. Zum Beispiel: [1, 2] ++
[3, 4] ergibt [1, 2, 3, 4].
concatLists :: [a] -> [a] -> [a]
concatLists xs ys = xs ++ ys


--Aufgabe 2: Listenfilterung
--Konzept: Filtern von Listen (filter)
--Aufgabe: Schreiben Sie eine Funktion filterEven :: [Int] -> [Int], die alle geraden Zahlen aus
einer Liste von ganzen Zahlen herausfiltert.
--Erklärung: Die filter-Funktion nimmt ein Prädikat (eine Funktion, die einen Booleschen Wert
zurückgibt) und eine Liste und gibt eine Liste der Elemente zurück, die das Prädikat erfüllen.
filterEven :: [Int] -> [Int]
filterEven xs = filter (isEven) xs
isEven :: Int -> Bool
isEven x = x 'mod' 2 == 0

--Aufgabe 3: Listenabbildung
--Konzept: Abbildung von Listen (map)
--Aufgabe: Schreiben Sie eine Funktion incrementAll :: [Int] -> [Int], die jede Zahl in einer Liste
um 1 erhöht.
```

```haskell
--Erklärung: Die map-Funktion wendet eine Funktion auf jedes Element einer Liste an und gibt eine neue Liste der Ergebnisse zurück.
incrementAll :: [Int] -> [Int]
incrementAll xs = map (+1) xs

--Aufgabe 6: Listenkomprehension
--Konzept: Listenkomprehension
--Aufgabe: Schreiben Sie eine Listenkomprehension, die die Quadrate aller ungeraden Zahlen in einer gegebenen Liste von ganzen Zahlen zurückgibt.
--Erklärung: Listenkomprehension ermöglicht das Erstellen von Listen durch Angabe eines Ausdrucks und optionaler Prädikate.
squaredList :: Num a => [a] -> [a]
squaredList xs = map (^2) (filter(odd)) xs

--Aufgabe 7: Listenzugriff
--Konzept: Zugriff auf Listenelemente
--Aufgabe: Schreiben Sie eine Funktion thirdElement :: [a] -> Maybe a, die das dritte Element einer Liste zurückgibt, falls vorhanden.
--Erklärung: Der Zugriff auf ein Element einer Liste erfolgt über den Indexoperator !!. Wenn der Index außerhalb der Grenzen liegt, gibt die Funktion Nothing zurück.
thirdElement :: [a] -> Maybe a
thirdElement xs = if length xs < 3 then Nothing else Just (xs !! 2)

--Aufgabe 8: Listenkonstruktion
--Konzept: Konstruktion von Listen
--Aufgabe: Schreiben Sie eine Funktion rangeList :: Int -> Int -> [Int], die eine Liste der ganzen Zahlen im Bereich von m bis n (einschließlich) erzeugt.
--Erklärung: Listen können durch Bereichsnotationen wie [m..n] konstruiert werden.
rangeList :: Int -> Int -> [Int]
rangeList l u = [l..u]

--Aufgabe 9: Listenlängenberechnung
--Konzept: Berechnung der Listenlänge (length)
--Aufgabe: Schreiben Sie eine Funktion lengthOfList :: [a] -> Int, die die Länge einer Liste berechnet.
--Erklärung: Die length-Funktion gibt die Anzahl der Elemente in einer Liste zurück.
lengthOfList :: [a] -> Int
lengthOfList xs = length xs

--Aufgabe 10: Listenreplikation
--Konzept: Replikation von Listen (replicate)
--Aufgabe: Schreiben Sie eine Funktion replicateElements :: Int -> a -> [a], die ein Element n-mal repliziert.
--Erklärung: Die replicate-Funktion erzeugt eine Liste, die ein gegebenes Element n-mal enthält.
replicateElements :: Int -> a -> [a]
replicateElements n x = replicate n x
```

FALTEN

```haskell
--Aufgabe 1: Summe einer Liste
--Konzept: Einfache Faltung mit foldl
--Aufgabe: Definiere eine Funktion sumList :: [Int] -> Int, die die Summe einer Liste von Ganzzahlen unter Verwendung von foldl berechnet.
--Erklärung: foldl führt eine linke Faltung auf einer Liste durch, beginnend mit einem Startwert und einem binären Operator.
sumList :: [Int]->Int
sumList x = foldl (+) 0


--Aufgabe 2: Produkt einer Liste
--Konzept: Einfache Faltung mit foldr
--Aufgabe: Definiere eine Funktion productList :: [Int] -> Int, die das Produkt einer Liste von Ganzzahlen unter Verwendung von foldr berechnet.
--Erklärung: foldr führt eine rechte Faltung auf einer Liste durch, beginnend mit einem Startwert und einem binären Operator.
productList :: [Int]-> Int
productList x = foldr (*) 1

--Aufgabe 3: Länge einer Liste
--Konzept: Faltung zur Berechnung der Länge
--Aufgabe: Schreibe eine Funktion lengthList :: [a] -> Int, die die Länge einer Liste unter Verwendung von foldl berechnet.
--Erklärung: Die Faltung kann verwendet werden, um die Elemente einer Liste zu zählen, indem man einen Zähler inkrementiert.
lengthList :: [a] -> Int
lengthList = foldl (\acc _ -> acc + 1) 0

--Aufgabe 4: Umkehrung einer Liste
--Konzept: Faltung zur Umkehrung
```

```haskell
--Aufgabe: Definiere eine Funktion reverseList :: [a] -> [a], die eine Liste unter Verwendung von foldl umkehrt.
--Erklärung: Listen können umgekehrt werden, indem man Elemente nacheinander an den Anfang einer neuen Liste anfügt.
reverseList :: [a] -> [a]
reverseList = foldl (flip (:)) []

--Aufgabe 5: Maximum einer Liste
--Konzept: Faltung zur Bestimmung des Maximums
--Aufgabe: Implementiere eine Funktion maximumList :: (Ord a) => [a] -> a, die das Maximum einer Liste von vergleichbaren Elementen unter Verwendung von foldr1 berechnet.
--Erklärung: foldr1 ist eine Variante von foldr, die keinen Startwert benötigt und den ersten Listeneintrag als Startwert verwendet.
maximumList :: (Ord a) => [a] -> a
maximumList = foldr1 max

--Aufgabe 6: Filter mit Faltung
--Konzept: Faltung zur Implementierung von filter
--Aufgabe: Schreibe eine Funktion filterFold :: (a -> Bool) -> [a] -> [a], die filter mit foldr nachbildet.
--Erklärung: foldr kann verwendet werden, um Elemente zu filtern, indem man nur diejenigen Elemente behält, die eine Bedingung erfüllen.
filterFold :: (a -> Bool) -> [a] -> [a]
filterFold p = foldr (\x acc -> if p x then x : acc else acc) []

--Aufgabe 7: Map mit Faltung
--Konzept: Faltung zur Implementierung von map
--Aufgabe: Definiere eine Funktion mapFold :: (a -> b) -> [a] -> [b], die map mit foldr nachbildet.
--Erklärung: foldr kann verwendet werden, um eine Funktion auf jedes Element einer Liste anzuwenden und die Ergebnisse zu sammeln.
--mapFold :: (a -> b) -> [a] -> [b]
--mapFold f = foldr (\x acc -> f x : acc) []

--Aufgabe 8: Konkatination von Listen
--Konzept: Faltung zur Verketten von Listen
--Aufgabe: Implementiere eine Funktion concatLists :: [[a]] -> [a], die eine Liste von Listen unter Verwendung von foldr zu einer einzelnen Liste konkateniert.
--Erklärung: foldr kann verwendet werden, um mehrere Listen zu einer einzigen Liste zu verbinden.
--concatLists :: [[a]] -> [a]
--concatLists = foldr (++) []


--Aufgabe 9: Faltungsfunktion für Binärbäume
--Konzept: Faltung auf benutzerdefinierten Datentypen
--Aufgabe: Definiere eine Funktion foldTree :: (b -> b -> b) -> (a -> b) -> BinaryTree a -> b, die eine Faltung auf einem binären Baum durchführt.
--Erklärung: Die Faltung kann auf benutzerdefinierte Datenstrukturen wie Bäume erweitert werden, indem man die Struktur rekursiv traversiert.
data BinaryTree a = Leaf a | Node (BinaryTree a) (BinaryTree a)
foldTree :: (b -> b -> b) -> (a -> b) -> BinaryTree a -> b
foldTree _ f (Leaf x) = f x
foldTree g f (Node left right) = g (foldTree g f left) (foldTree g f right)

--Aufgabe 10: Links- und Rechtsfaltung vergleichen
--Konzept: Unterschied zwischen foldl und foldr
--Aufgabe: Schreibe eine Funktion compareFolds :: [a] -> (b -> a -> b) -> b -> (b, b), die dieselbe Liste sowohl mit foldl als auch mit foldr faltet und die Ergebnisse zurückgibt.
--Erklärung: Das Verständnis der Unterschiede zwischen linker und rechter Faltung ist entscheidend, insbesondere in Bezug auf Reihenfolge und Assoziativität der Operationen.
compareFolds :: [a] -> (b -> a -> b) -> b -> (b, b)
compareFolds xs f init = (foldl f init xs, foldr f init xs)


-- reverse nur mit fold
-- foldl ist eine Funktion, die eine Liste von Werten auf einen einzelnen Wert reduziert.
-- \ ist eine anonyme Funktion, die zwei Argumente nimmt und die Summe dieser Argumente zurückgibt.
-- acc ist der Akkumulator, der den Wert der Reduktion speichert.
-- x ist das aktuelle Element der Liste.
-- x : acc fügt das aktuelle Element x an den Anfang des Akkumulators acc an.
-- foldl (\acc x -> x : acc) [] xs wendet die Funktion auf die Liste xs an, wobei der Akkumulator zu Beginn leer ist.
-- [] ist der Startwert des Akkumulators.
reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []
-- reverse' [1, 2, 3, 4]
-- Schrittweise Anwendung des Folds:
-- 1. Initialer Akkumulator: []
-- 2. Nach dem ersten Element (1): 1 : [] = [1]
-- 3. Nach dem zweiten Element (2): 2 : [1] = [2, 1]
-- 4. Nach dem dritten Element (3): 3 : [2, 1] = [3, 2, 1]
-- 5. Nach dem vierten Element (4): 4 : [3, 2, 1] = [4, 3, 2, 1]
```

```haskell
-- Ergebnis: [4, 3, 2, 1]

-- deepReverse nur mit reverse und map
-- map wendet eine Funktion auf jedes Element einer Liste an.
-- reverse' wendet reverse auf eine Liste an.
-- . ist die Funktionskomposition, die zwei Funktionen verknüpft.
-- deepReverse wendet reverse' auf jede innere Liste an und kehrt die Reihenfolge der äußeren Liste
um.
deepReverse :: [[a]] -> [[a]]
deepReverse = map reverse' . reverse'
-- deepReverse ["hallo", "welt"]
-- Schrittweise Anwendung:
-- 1. Äußere Liste umkehren: reverse' ["hallo", "welt"] = ["welt", "hallo"]
-- 2. Jede innere Liste umkehren: map reverse' ["welt", "hallo"] = ["tlew", "ollah"]
-- Ergebnis: ["tlew", "ollah"]

-- die beiden Komponenten der Tupel addiert und alle Teilergebnisse dann multipliziert
f :: [(Int, Int)] -> Int
f = foldl (\acc (x, y) -> acc * (x + y)) 1
--f [(1, 2), (3, 4)]
-- Schrittweise Anwendung des Folds:
-- 1. Initialer Akkumulator: 1
-- 2. Nach dem ersten Element (1, 2): 1 * (1 + 2) = 1 * 3 = 3
-- 3. Nach dem zweiten Element (3, 4): 3 * (3 + 4) = 3 * 7 = 21
-- Ergebnis: 21

-- Just  ist ein Konstruktor für den Maybe-Datentyp, der einen Wert enthält.
-- Just dient dazu, einen Wert in den Maybe-Datentyp zu verpacken.
-- foldl dient dazu, eine Liste von Werten auf einen einzelnen Wert zu reduzieren.
-- (*) ist der Multiplikationsoperator.
-- 1 ist der Startwert des Akkumulators.
-- [1..n] ist eine Liste von 1 bis n.
fac :: Integer -> Maybe Integer
fac n
  | n < 0     = Nothing
  | otherwise = Just $ foldl (*) 1 [1..n]
--fac 5
-- foldl (*) 1 [1, 2, 3, 4, 5]
-- Schrittweise Anwendung des Folds:
-- 1. Initialer Akkumulator: 1
-- 2. Nach dem ersten Element (1): 1 * 1 = 1
-- 3. Nach dem zweiten Element (2): 1 * 2 = 2
-- 4. Nach dem dritten Element (3): 2 * 3 = 6
-- 5. Nach dem vierten Element (4): 6 * 4 = 24
-- 6. Nach dem fünften Element (5): 24 * 5 = 120
-- Ergebnis: 120
-- Rückgabe: Just 120
```