Aufgabe 1 Typsignaturen (1+1+1 Punkte)

a) Geben Sie in Haskell-Notation einen Lambda-Ausdruck an, der den Typ

```
(b -> c) -> (a -> b) -> a -> c
```

hat. Benutzen Sie nicht den Operator `(.)` zur Funktionskomposition.

```
\f g x -> f (g x)
```

b) Geben Sie die Haskell-Typsignatur einer beliebigen einstelligen Funktion `p` mit polymorphem Typ an. Verwenden Sie dabei ausschließlich den polymorphen Typ `a`.

```
p :: a -> a
```

p: (a) → a

c) Geben Sie die Haskell-Typsignatur einer beliebigen Funktion `u` in uncurried Form an, deren curried Form die drei Argumente `a`, `b` und `c` und das erste Element als Rückgabewert hat.

```
u :: (c, a, b) -> c
```

Curried ↓

```
u :: c -> a -> b -> c
u x y z = x
```

curried

```
add3 :: Int -> Int -> Int -> Int
add3 x y z = x + y + z
```

add3 uncurried :: (Int, Int, Int) → Int

add3 uncurried (5, 4, 100)

```
add10 = add3 7 13
add10 100 =
= 7 + 13 + 100
```

Aufgabe 2 Rekursion (1+1+1 Punkte)

Das innere Produkt zweier Vektoren $\vec{x}, \vec{y}$ ist definiert als

$$\sum_{i=1}^{n} x_i \cdot y_i.$$

Ein Vektor soll in Haskell durch eine nicht-leere endliche Liste repräsentiert werden.

a) Schreiben Sie eine rekursive, aber nicht endrekursive Haskell-Funktion

```
iprod :: Num a => [a] -> [a] -> a
```

die das innere Produkt zweier Vektoren ausgibt. Dabei soll bei unterschiedlicher Länge der Vektoren der kürzere Vektor so behandelt werden, als wären seine fehlenden Werte gleich null. Verwenden Sie dafür von den Listenfunktionen nur den Listenkonstruktor `(:)`; **`head`** und **`tail`** sind nicht erlaubt.

```
iprod :: Num a => [a] -> [a] -> a
iprod xs [] = 0
iprod [] ys = 0
iprod (x:xs) (y:ys) = x * y + iprod xs ys
```

frägt, ob eine Liste leer ist

```
null :: [a] -> Bool
null [] = True
null _  = False
```

b) Geben Sie eine endrekursive Haskell-Funktion **`iprod'`** an, die sich wie **`iprod`** verhält. Sie dürfen dafür die Listenfunktionen `(:)`, **`null`**, **`head`** und **`tail`** verwenden.

```
iprod' :: Num a => [a] -> [a] -> a
iprod' xs ys = ip xs ys 0
  where ip xs ys akk =
          if null xs || null ys
          then akk
          else ip (tail xs) (tail ys) (akk+(head xs)*(head ys))
```

```
head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs
```

Skalarprodukt

$$\begin{pmatrix}5\\3\\7\end{pmatrix} \cdot \begin{pmatrix}2\\1\\3\end{pmatrix} = 10 + 3 + 21 = 34$$

$$\begin{pmatrix}5\\3\\7\\7\\100\end{pmatrix} \cdot \begin{pmatrix}2\\1\\3\\0\\0\end{pmatrix} = \begin{pmatrix}5\\3\\7\\7\\100\end{pmatrix} \cdot \begin{pmatrix}2\\1\\3\\0\\0\end{pmatrix} = 34$$

$$\begin{pmatrix}5\\3\\7\end{pmatrix} \cdot \begin{pmatrix}2\\1\\3\end{pmatrix} = 5 \cdot 2 + \begin{pmatrix}3\\7\end{pmatrix} \cdot \begin{pmatrix}1\\3\end{pmatrix}$$

c) Realisieren Sie unter Verwendung der Listenfunktionen **zip** und entweder **foldl** oder **foldr** einen Lambda-Ausdruck in Haskell-Notation, den Sie **iprod''** nennen und der sich sich wie **iprod** und **iprod'** verhält.

```
iprod'' :: Num a => [a] -> [a] -> a

iprod'' = \xs ys -> _________ (\(a, b) r -> r + _________*_________)
                                0 (zip _________ _________)
```

zip ["Hallo", "Word"] [4, 100]

[("Hallo", 4), ("Wold", 100)]

```
zip :: [a] -> [b] -> [(a,b)]
zip [ ]  ys    = [ ]
zip  xs  [ ]   = [ ]
zip (x:xs) (y:ys) = [(x,y)] ++ zip xs ys
```

oder

```
(x,y) : zip xs ys
```

```
foldr :: (a -> b -> b) -> [a] -> b -> b
foldr f  [ ]  akk = akk
foldr f (x:xs) akk = foldr f xs (f x akk)
```