null :: [a] → Bool
null [ ] = True
null _ = False

tail :: [a] → [a]
tail (x:xs) ↦ xs

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+0
          else ip (tail xs) (tail ys) (akk+(head xs)*(head ys))
```

ip [2,3,4] [3,7,5] 0 =
= ip [3,4] [7,5] (0 + 2·3)
= ip [4] [5] (6 + 3·7)
= ip [ ] [ ] (6+3·7+4·5)
= 6+3·7+4·5+0

[3,7,1] ∘ [2,4,5]
⇓
[(3,2),(7,4),(1,5)]
⇓

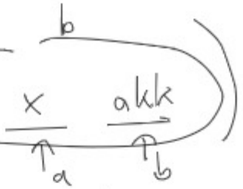
c) Realisieren Sie unter Verwendung der Listenfunktionen `zip` und entweder `foldl` oder `foldr` einen Lambda-Ausdruck in Haskell-Notation, den Sie `iprod''` nennen und der sich sich wie `iprod` und `iprod'` verhält.

```
iprod'' :: Num a => [a] -> [a] -> a
iprod'' = \xs ys -> foldl (\(a, b) r -> r + a * b) 0 (zip xs ys)
```

zip [1,2] ["H", "a", "b"] = [(1,"H"),(2,"a")]

zip :: [a] → [b] → [(a,b)]
zip [ ] _ = [ ]
zip _ [ ] = [ ]
zip (x:xs) (y:ys) = [(x,y)] ++ zip xs ys
~~zip xs ys = (head xs, head ys) : zip (tail xs) (tail ys)~~

foldl :: (a → b → b) → b → [a] → b
foldl f akk [ ] = akk
foldl f akk (x:xs) = foldl f (f x akk) xs



Beispiel: foldl (+) 0 [1,4,7] =
= foldl (+) (1+0) [4,7] =
= foldl (+) (4+1+0) [7] =
= foldl (+) (7+4+1+0) [ ] = 7+4+1+0

miro

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

(x : a, foldr f akk xs : b)

foldr (+) 0 [1,4,7] =

= (+) 1 (foldr (+) 0 [4,7])

= (+) 1 ((+) 4 foldr (+) 0 [7])

= (+) 1 ((+) 4 ((+) 7 (foldr (+) 0 [])))

foldr (+) 0 [] = 0

= (1 + (4 + (7 + 0)))

→ 62

miro

foldr (\x akk → akk + 1) 0 [1,3,7] = 3

reverse :: [a] → [a]

reverse [1,3,7] = [7,3,1]

~~reverse [] = []~~

~~reverse (x : xs) = reverse xs ++ [x]~~

reverse xs = foldr (\x akk → ~~[x] ++ akk~~ akk ++ [x]) [] xs

= f

foldr f [] [1,3,7]

= f 1 (foldr f [] [3,7])

= (foldr f [] [3,7]) ++ [1]

miro

reverse xs = foldl (\x akk → [x] ++ akk) [ ] xs

= f

foldl f [ ] [1,3,7]

= foldl f (f 1 [ ]) [3,7]

= foldl f [1] [3,7]

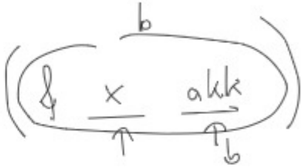
= foldl f (f 3 [1]) [7]

= foldl f ([3,1]) [7]

= foldl f (f 7 [3,1]) [ ]

= foldl f [7,3,1] [ ] = [3,7,1]

foldl :: (a → b → b) → b → [a] → b
foldl f akk [ ] = akk
foldl f akk (x:xs) = foldl f (f x akk) xs



miro

c) Gegeben seien die folgenden Haskell-Definitionen:

Erst werden die Argumente ausgewertet und dann die Funktion

```
qsum = \a b -> a*a + b*b
quad = \n -> n*n
```

- Wird bei einer applikativen Auswertung des Ausdrucks `quad (qsum 3 4)` die Summe 9+16 genau ein Mal berechnet?
  ☒ Ja ☐ Nein
- Wird bei einer normalen Auswertung des Ausdrucks `quad (qsum 3 4)` die Summe 9+16 genau ein Mal berechnet?
  ☐ Ja ☒ Nein
- Wird bei einer verzögerten Auswertung des Ausdrucks `quad (qsum 3 4)` die Summe 9+16 genau ein Mal berechnet?
  ☒ Ja ☐ Nein

Erst wird die Funktion ausgewertet und dann die ARgument

qsum 4 5 = 16 + 25
= 41
quad 9 = 81

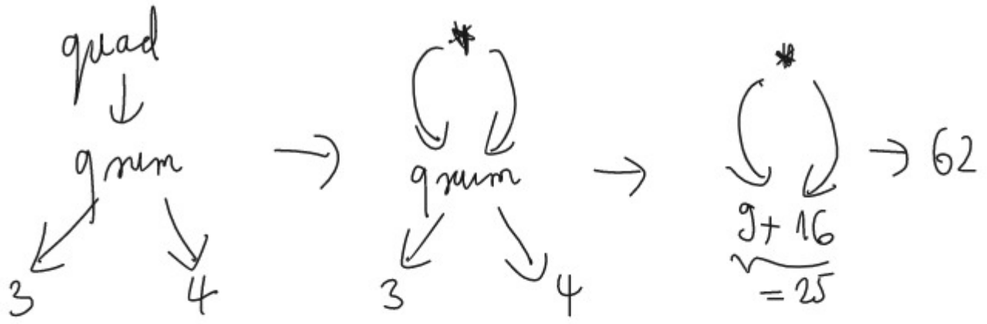
foldr
foldr
foldr

applikativ
quad (qsum 3 4) =
= quad (3·3 + 4·4) = quad (25) = 25·25 = 625

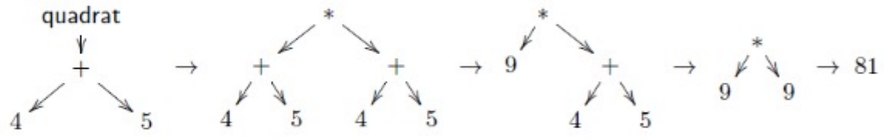
normal
quad (qsum 3 4) =
= (qsum 3 4) · (qsum 3 4) =
= (9 + 16) · (9 + 16)
= 25 · (9 + 16) =
= 25 · 25 = 625

verzögert

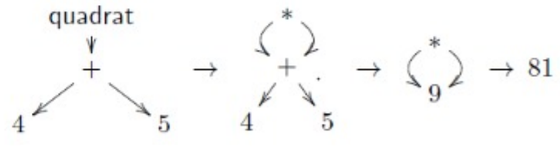
Beispiel in gerichteter-Graph-Darstellung



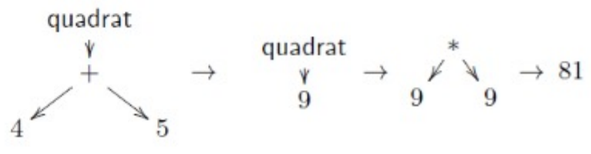
Normale Reihenfolge:



Verzögerte Reihenfolge:



Applikative Reihenfolge:



miro