```
--1)
--Sorry Axel, ich habe mal die innere List-Comprehension zuerst gemacht,
--sollte eigentlich nicht so sein, aber es geht trotzdem und ich habe
--den Kram verstanden. Ruhe jetzt ;D

innere = [[1..u] | u <- [3..30], even u]

step0 = [x*3 | z <- innere, x <- z, odd x]

step1 = let ok u = [[1..u] | even u]
 in concatMap ok [3..30]

step2 = let ok u = if even u then [[1..u]] else []
 in concatMap ok [3..30]

step3 = [x*3 | z <- step2, x <- z, odd x]

step4 = let ok z = [x*3 | x <- z, odd x]
 in concatMap ok step2

step5 = let ok z = let ok' x = [x*3 | odd x]
                   in concatMap ok' z
        in concatMap ok step2

step6 = let ok z = let ok' x = if odd x then [x*3] else []
                   in concatMap ok' z
        in concatMap ok step2

--step6 == step0 evaluiert zu True :)


--2)
data Expr a =
 Var a -- x = Var "x"
 | App (Expr a) (Expr a) -- (e1 e2) = App e1 e2
 | Lam a (Expr a) -- \x.e = Lam "x" e
 | ListCons (Expr a) (Expr a) -- a:as = ListCons a as
 | ListNil -- [] = ListNil
 | BoolTrue -- True = BoolTrue
 | BoolFalse -- False = BoolFalse
 | CaseList (Expr a) (Expr a) (a,a,Expr a)-- case_List e of {[] -> e1; (x:xs) -> e2)
-- = CaseList e e1 ("x",xs",e2)
 | CaseBool (Expr a) (Expr a) (Expr a) -- case_Bool e of {True -> e1; False -> e2)
-- = CaseBool e e1 e2

k x = drop 1 $ reverse $ drop 1 $ reverse x --Die Funktion hieß mal "keineHochkomma" aber das
war mir zu lange zu tippen
--es blieb nur das k übrig.

third (a, b, c) = c

showLambda (Var a) = k $ show a
```

Ich schieb es mal darauf das ich so krank war am WE !-

$\Sigma = 41{,}5/50$

ok _ = [] ok' _ = []
(step 2) ok _ = []

95/10

↳ Dies gilt aber nur, wenn a :: String, sonst kickst du ein paar wichtige Zeichen !!

```
showLambda (App a b) = "(" ++ (showLambda a) ++ " " ++ (showLambda b) ++ ")"
showLambda (Lam a b) = "\\" ++ (k $ show a) ++ " -> " ++ (showLambda b)
showLambda (ListCons a b) = (showLambda a) ++ ":" ++ (showLambda b)
showLambda (ListNil) = "[]"
showLambda (BoolTrue) = "True"
showLambda (BoolFalse) = "False"
showLambda (CaseList e e1 e2) = "case_List " ++ (showLambda e) ++ " of {[] -> " ++
(showLambda e1) ++ "; (x:xs) -> " ++ (showLambda $ third e2) ++ "}"
showLambda (CaseBool e e1 e2) = "case_Bool " ++ (showLambda e) ++ " of {True -> " ++
(showLambda e1) ++ "; False -> " ++ (showLambda e2) ++ "}"

instance Show a => Show (Expr a) where
  show = showLambda

beispiel4 =
 Lam "x"
   (CaseList
     (Var "x")
     (Var "x")
     ("y","ys",(Var "ys")))
--jo sieht alles gut aus

--3)

data BBaum a = BBlatt Int -- Blatt mit Markierung
  | BKnoten Int (BBaum a) (BBaum a) -- Markierung, linker u. rechter Teilbaum
  deriving(Show)
data NBaum a = NBlatt Int -- Blatt mit Markierung
  | NKnoten Int [NBaum a] -- Markierung und Liste der Kinder
  deriving(Show)

bspBBaum = BKnoten 1 (BKnoten 2 (BBlatt 3) (BBlatt 4)) (BKnoten 5 (BBlatt 6) (BBlatt 7))
bspNBaum = NKnoten 1 [NKnoten 2 [NBlatt 3,NBlatt 4, NBlatt 5] ,NKnoten 6 [NBlatt 7, NKnoten
8 [NBlatt 9, NBlatt 10]]]

--a)
class IstBaum a where
  teilbaeume :: a -> [a]
  istBlatt :: a -> Bool

instance IstBaum (BBaum a) where
  teilbaeume (BBlatt a) = []
  teilbaeume (BKnoten a s d) = [s, d]
  istBlatt (BBlatt a) = True
  istBlatt (BKnoten a s d) = False

instance IstBaum (NBaum a) where
  teilbaeume (NKnoten a s) = s
  teilbaeume (NBlatt a) = []
  istBlatt (NKnoten a s) = False
  istBlatt (NBlatt a) = True
```

```
--b)
class (IstBaum a) => IstMarkierterBaum a where
  markierung :: a -> Int
  knoten :: a -> [Int]
  kanten :: a -> [(Int, Int)]

--c)
instance IstMarkierterBaum (BBaum a) where
  markierung (BBlatt a) = a
  markierung (BKnoten a s d) = a
  knoten (BBlatt a) = [a]
  knoten (BKnoten a s d) = [a] ++ knoten s ++ knoten d
  kanten (BBlatt a) = []
  kanten (BKnoten a s d) = [(a, markierung s)] ++ [(a, markierung d)] ++ kanten s ++ kanten d

instance IstMarkierterBaum (NBaum a) where
  markierung (NBlatt a) = a
  markierung (NKnoten a s) = a
  knoten (NBlatt a) = [a]
  knoten (NKnoten a s) = [a] ++ (concat $ map knoten s)
  kanten (NBlatt a) = []
  kanten (NKnoten a s) = [(x, y) | x <- [a], y <- map markierung s] ++ (concat $ map kanten s)

--habe alles gestestet, es kommt stets das raus, was auf dem Arbeitsblatt angegeben war :))
```

Nicht jede Markierung ist ein Int

(a b) → b hier ist b die Markierung

Wo sind die Default-Implementierungen?

✓

✓

✓

✓

✓

Das impliziert leider nicht die Richtigkeit :-

✓

Default

a) 8/8

b) 7/12

c) 10/10

25/30