Fork (Fork (Tip 2) (Fork (Tip 3) (Tip 5)))
(Fork (Tip 1) (Tip 2)

—) **F**

Para qualquer α : Type, definimos

[2]++[3]++[5]

```
data Tree a                                  data Dir        type Path = List Dir
  Tip  : a → Tree a                            L : Dir
  Fork : Tree a → Tree a → Tree a              R : Dir
```

(12) **F1.** Defina as funções:

```
depth   : Tree a → Nat          flatten : Tree a → List a
ntips   : Tree a → Nat          mirror  : Tree a → Tree a
nforks  : Tree a → Nat          fetch   : Tree a → Path → Maybe a
```

DEFINIÇÕES.

```
ntips (Fork T1 T2) = ntipsT1 + ntipsT2
ntips (Tip x) = 1
nforks (Fork T1 T2) = 1 + nforksT1 + nforksT2
nforks (Tip x) = 0
flatten (Tip a) = [a]
flatten (Fork T1 T2) = flattenT1 ++ flattenT2
mirror (Tip a) = Tip a
mirror (Fork T1 T2) = Fork (mirrorT1) (mirrorT2)
                                  2          1
```

(12) **F2.** Defina uma versão eficiente de *flatten* usando uma função auxiliar *flatten'*.

Dica: Use indução.

DEFINIÇÃO.

(4) **F3.** Enuncie uma equação interessante sobre *flatten* e *mirror*.

RESPOSTA.

```
flatten . mirror = reverse . flatten
```

```
fetch (Tip a) [] = Just a
fetch (Fork t1 t2) (d:ds) = match d with
                              L = fetch t1 ds
                              R = fetch t2 ds
fetch _ _ = Nothing
depth (Tip a) = 0
depth (Fork t1 t2) = if m ≥ n then m else n
                     where m = 1 + depth t1
                           n = 1 + depth t2
```

max m n

poderia cochuveirinhar

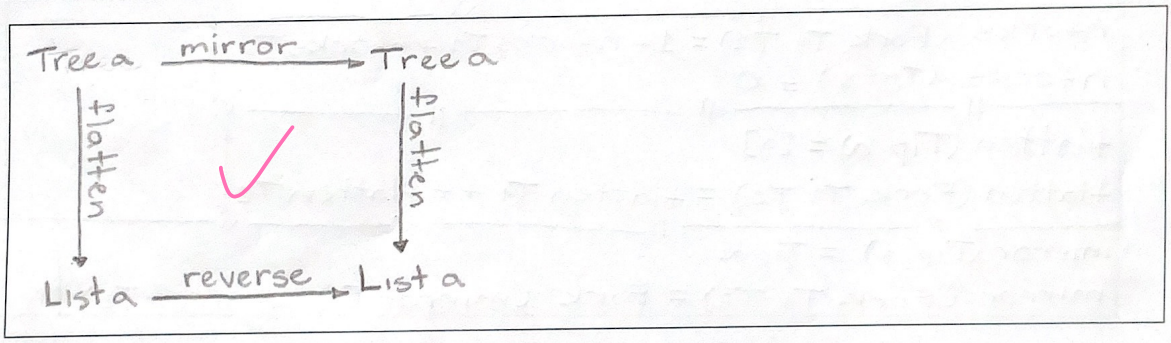
(16) **F4.** Enuncie e demonstre uma equação interessante sobre *ntips* e *nforks*.
RESPOSTA.

(∀T : Tree)[ntips T = nforks T + 1]
Por indução em T.
BASE:
ntips (Tip a) = 1 [ntips.2]
nforks (Tip a) + 1 = 0 + 1 [nforks.2]
= 1 [(+).1]

PASSO INDUTIVO:
Suponha que a equação é válida para t1, t2 : Tree. (HI)
Queremos mostrar que o mesmo é válido para Fork t1 t2.

RESTO NO RASCUNHO

(10) **F5.** Desenhe os diagramas comutativos que correspondem aos teoremas **F3** e **F4**.
RESPOSTA.



(8) **F6.** Modifique o Tree para representar arvores que carregam informações não apenas nas folhas, mas nos nós também, mantendo a natureza binária.
RESPOSTA.

data Tree a B
Tip :: a → Tree a
Fork :: ~~a~~ β → Tree a β → Tree a β → Tree a β

(8) **F7.** Qual seria um tipo razoável para a *flatten* desse tipo de arvores?
Defina tal *flatten* (não se preocupe com eficiência).
RESPOSTA.

```
flatten :: Tree a → List a
flatten (Tip x) = [x]
flatten (Fork x T1 T2) = flatten T1 ++ [x] ++
                         flatten t2
```

RASCUNHO

F4. T = Fork t1 t2

Calculemos:

ntips T = ntips (Fork t1 t2) [Def]

= ntips t1 + ntips t2 [ntips.1]

= nforks t1 + 1 + nforks t2 + 1 [HI]

= 1 + nforks t1 + nforks t2 + 1 [(+)-com]

= nforks (Fork t1 t2) + 1 [nforks.1]

= nforks T + 1 // [Def.

(—5) E

Considere o tipo de expressões aritméticas envolvendo apenas números e divisões inteiras:

```
data Expr
  Val : Int → Expr
  Div : Expr → Expr → Expr
```

(4) **E1.** Dada a função *div* que retorna o quociente explique qual o problema da função seguinte:

```
eval : Expr → Int
eval (Val n) = n
eval (Div u v) = div (eval u) (eval v)
```

RESPOSTA.

Caso v seja 0, teremos uma divisão por 0. ✓

(8) **E2.** Defina uma versão segura (total) de *div* (chame de *safediv*).
DEFINIÇÃO.

```
safediv :: Int → Int → Maybe Int
safediv m n = if n == 0 then Nothing
              else Just (div m n)
```

✓

(melhor pattern-matcher com 0

(24) **E3.** Usando tua função *safediv* defina uma melhor *eval* : Expr → ?.
DEFINIÇÃO.

```
eval : Expr → Maybe Int
eval (Val n) = n
eval (Div u v) = safediv (eval u) (eval v)
```

Type Error!

(12) **E4.** Um aluno—talvez não o melhor da turma—respondeu na **E3** assim:

```
eval : Expr → Maybe Int
eval (Val n)   = pure n
eval (Div u v) = pure safediv <*> eval u <*> eval v
```

Compila? Se não, explique o porquê. Se sim, explique porque seria melhor evitar esta definição.
RESPOSTA.

Não compila, pois type error.
mas cadê?

Só isso mesmo.

F

Para qualquer α : Type, definimos

```
data Tree a                                   data Dir        type Path = List Dir
  Tip  : a → Tree a                             L : Dir
  Fork : Tree a → Tree a → Tree a               R : Dir
```

(12) **F1.** Defina as funções:

```
depth   : Tree a → Nat          flatten : Tree a → List a
ntips   : Tree a → Nat          mirror  : Tree a → Tree a
nforks  : Tree a → Nat          fetch   : Tree a → Path → Maybe a
```

DEFINIÇÕES.

```
depth (Tip _) = S 0
depth (Fork x y) = IF (depth x)>(depth y)
                   then S(depth x)
                   else S(depth y)

ntips (Tip _) = S0
ntips (Fork x y) = (ntips x) + (ntips y)

nForks (Tip _) = 0
nforks (Fork x y) = S((nForks x) + (nforks y))
```

```
Flatten (Tip x) = [x]
Flatten (Fork x y) = (Flatten x) ++ (Flatten y)

mirror (Tip x) = Tip x
mirror (Fork x y) = Fork (mirror y) (mirror x)

Fetch (Tip x) [] = Just x
Fetch (Fork x y) (p:ps) = cases p of
                            L → Fetch x ps
                            R → Fetch y ps
Fetch _ _ = Nothing
```

(12) **F2.** Defina uma versão eficiente de *flatten* usando uma função auxiliar *flatten′*.

Dica: Use indução.

DEFINIÇÃO.

(4) **F3.** Enuncie uma equação interessante sobre *flatten* e *mirror*.

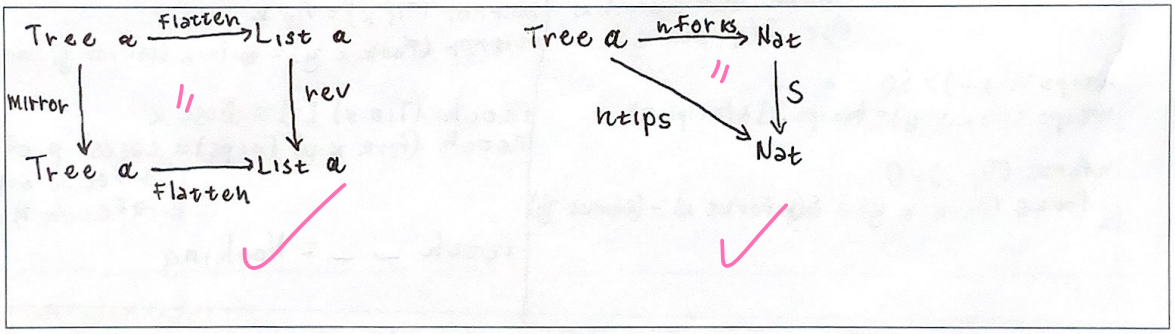
RESPOSTA.

```
Flatten (mirror t) = rev (Flatten t)
```

(16) **F4.** Enuncie e demonstre uma equação interessante sobre *ntips* e *nforks*.
RESPOSTA.

ntips t = S(nforks t)
Por indução no t.
S(nforks (tip w)) = S0 [nforks.1]
ntips (tip w) = S 0 [ntips.1]
Passo indutivo
ntips (fork u v) = (ntips u) + (ntips v) [ntips.2]
S(nforks (fork u v)) = S(S((nforks u) + (nforks v))) [nforks.2]
= S(nforks u) + S(nforks v) [P.I.]
= S(S((nforks u) + (nforks v)))
= S(S((nforks u) + (nforks v)))

(10) **F5.** Desenhe os diagramas comutativos que correspondem aos teoremas **F3** e **F4**.
RESPOSTA.



(8) **F6.** Modifique o Tree para representar arvores que carregam informações não apenas nas folhas, mas nos nós também, mantendo a natureza binária.
RESPOSTA.

(8) **F7.** Qual seria um tipo razoável para a *flatten* desse tipo de arvores? Defina tal *flatten* (não se preocupe com eficiência).
RESPOSTA.

E

Considere o tipo de expressões aritméticas envolvendo apenas números e divisões inteiras:

```
data Expr
  Val : Int → Expr
  Div : Expr → Expr → Expr
```

(4) **E1.** Dada a função *div* que retorna o quociente explique qual o problema da função seguinte:

```
eval : Expr → Int
eval (Val n) = n
eval (Div u v) = div (eval u) (eval v)
```

RESPOSTA.

Chegar em uma divisão por 0.

(8) **E2.** Defina uma versão segura (total) de *div* (chame de *safediv*).
DEFINIÇÃO.

```
safediv : Int → Int → Maybe Int
sd _ 0 = Nothing
sd x y = div x y
```

(24) **E3.** Usando tua função *safediv* defina uma melhor *eval* : Expr → ?.
DEFINIÇÃO.

(12) **E4.** Um aluno—talvez não o melhor da turma—respondeu na **E3** assim:

```
eval : Expr → Maybe Int
eval (Val n)   = pure n
eval (Div u v) = pure safediv <*> eval u <*> eval v
```

Compila? Se não, explique o porquê. Se sim, explique porque seria melhor evitar esta definição.
RESPOSTA.

Compila.

Só isso mesmo.

(70) **F**

Para qualquer α : Type, definimos

```
data Tree a                                   data Dir        type Path = List Dir
  Tip  : a → Tree a                             L : Dir
  Fork : Tree a → Tree a → Tree a               R : Dir
```

(12) **F1.** Defina as funções:

```
depth   : Tree a → Nat          flatten : Tree a → List a
ntips   : Tree a → Nat          mirror  : Tree a → Tree a
nforks  : Tree a → Nat          fetch   : Tree a → Path → Maybe a
```

use pattern matching!

DEFINIÇÕES.

```
depth (Tip a) = 0
depth (t 'fork' t') = 1 + max (depth t) (depth t')

ntips (Tip a) = 1
ntips (t 'fork' t') = ntips t + ntips t'

nforks (Tip a) = 0
nforks (t 'fork' t') = if ntips t == 1 then nforks t' else nforks t + nforks t'

flatten (Tip a) = [a]
flatten (t 'fork' t') = flatten t ++ flatten t'

mirror (Tip a) = Tip a
mirror (t 'fork' t') = mirror t' 'fork' mirror t

fetch (Tip a) [] = Just a
fetch (Tip a) (x : xs) = Nothing
fetch (t 'fork' t') [] = Nothing
fetch (t 'fork' t') (x : xs) = if x == L then fetch t xs else fetch t' xs
```

por que olhar nisso?

(12) **F2.** Defina uma versão eficiente de *flatten* usando uma função auxiliar *flatten′*.

Dica: Use indução.

DEFINIÇÃO.

(4) **F3.** Enuncie uma equação interessante sobre *flatten* e *mirror*.

RESPOSTA.

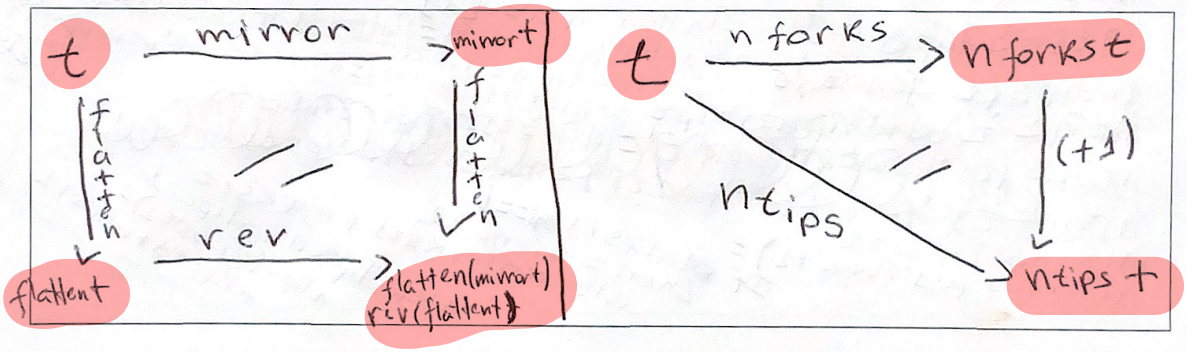
```
flatten (mirror t) = rev (flatten t)
```

(16) **F4.** Enuncie e demonstre uma equação interessante sobre *ntips* e *nforks*.
RESPOSTA.

ntips t = (nforks t) + 1

tipos aqui!!

(10) **F5.** Desenhe os diagramas comutativos que correspondem aos teoremas **F3** e **F4**.
RESPOSTA.



(8) **F6.** Modifique o Tree para representar arvores que carregam informações não apenas nas folhas, mas nos nós também, mantendo a natureza binária.
RESPOSTA.

```
data Tree a b:
  a :type  b :type
  Tip : a -> Tree a b
  Fork : Tree a b -> Tree a b -> b -> Tree a b
```

(8) **F7.** Qual seria um tipo razoável para a *flatten* desse tipo de arvores?
Defina tal *flatten* (não se preocupe com eficiência).
RESPOSTA.

```
flatten : Tree a b -> List (either a b)
flatten (Tip a) = [left a]
flatten (Fork t1 t2 u) = flatten t1 ++ [right b] ++ flatten 2
```

(48) **E**

Considere o tipo de expressões aritméticas envolvendo apenas números e divisões inteiras:

```
data Expr
  Val : Int → Expr
  Div : Expr → Expr → Expr
```

(4) **E1.** Dada a função *div* que retorna o quociente explique qual o problema da função seguinte:

```
eval : Expr → Int
eval (Val n) = n
eval (Div u v) = div (eval u) (eval v)
```

RESPOSTA.

a função assume um inteiro como resposta a uma possível divisão por zero.

(8) **E2.** Defina uma versão segura (total) de *div* (chame de *safediv*).
DEFINIÇÃO.

```
Safediv : Expr -> Expr -> Maybe Expr
Safediv [illegible] _ 0 = Nothing
Safediv n m = Just (div n m)
```

(24) **E3.** Usando tua função *safediv* defina uma melhor *eval* : Expr → ?.
DEFINIÇÃO.

```
eval : Expr -> maybe int
eval (Val n) = Just n
eval ([illegible] div u v) = safediv (fromJust u) (fromJust v)
```

type error

(12) **E4.** Um aluno—talvez não o melhor da turma—respondeu na **E3** assim:

```
eval : Expr → Maybe Int
eval (Val n)   = pure n
eval (Div u v) = pure safediv <*> eval u <*> eval v
```

Compila? Se não, explique o porquê. Se sim, explique porque seria melhor evitar esta definição.
RESPOSTA.

Sim, compila. todavia seria interessante não utilizar essa definição porque o pure está definido como identidade de forma distinta da definição correspondente ao splat. Splat=temporal. Identidade=ambíguo

Só isso mesmo.

(22) G

(por que mul x y em vez de x · y e add x y e x + y ?)

(10) **G1.** Complete as equações seguintes **com algo interessante:**[3]

✗ `map f (flatten t)` = map f (reverse $ flatten (mirror t))

✓ `sum (replicate n x)` = mul n x

✓ `sum (map (+ k) ns)` = add (mul K (len ns)) (sum ns)

✗ `sum (map (· k) ns)` = add (Prod (replicate K (len ns)) (sum ns)

✗ `sorted (map (+ k) ns)` = not (sorted (reverse ns))

(12) **G2.** Escolha **uma** das equações de **G1** para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto se foram definidas em outras questões desta prova.

DEFINIÇÕES.

```
sum :: [Nat] -> Nat
sum [] = 0
sum [n:ns] = n + (sum ns)
```

(isso é uma lista com um único membro)

```
repl :: Nat -> Nat -> [Nat]
repl 0 m = []
repl (Sn) m = m : (repl n m)
```

```
mul :: Num n => n -> n -> n
mul (Sn) m = m + (mul n m)
mul 0 _ = 0
mul _ 0 = 0
```

→ 3 equações para definir a multiplicação? ::

DEMONSTRAÇÃO DE sum (repl n x) = mul n x .

indução

-- Base: n = 0

Seja x.

Calc: sum (repl 0 x)
= sum [] [repl.1]
= 0. [sum.1]
mul 0 x
= 0. [mul.2]

-- P.I.: n = SK

Seja x.

Calc: sum (repl (SK) x)
= sum (x : repl K x) [repl.2]
= x + sum (repl K x) [sum.2]
= x + (mul K x). [Hip. Ind.]

mul (SK) x
= x + (mul K x). [mul.2]

Hip. Ind.: sum (repl K x) = mul K x

[3] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

F

Para qualquer α : Type, definimos

```
data Tree a                                   data Dir        type Path = List Dir
  Tip  : a → Tree a                             L : Dir
  Fork : Tree a → Tree a → Tree a               R : Dir
```

(12) **F1.** Defina as funções:

```
depth   : Tree a → Nat          flatten : Tree a → List a
ntips   : Tree a → Nat          mirror  : Tree a → Tree a
nforks  : Tree a → Nat          fetch   : Tree a → Path → Maybe a
```

DEFINIÇÕES.

```
ntips (Tip a) = 1
ntips (Fork l r) = ntips l + ntips r

mirror (Tip a) = Tip a
mirror (Fork l r) = Fork (mirror r) (mirror l)

nforks (Tip a) = 0
nforks (Fork l r) = 1 + nforks l + nforks r

depth (Tip a) = 0
depth (Fork l r) = max (1 + depth l) (1 + depth r)

flatten (Tip a) = [a]
flatten (Fork l r) = flatten l ++ flatten r
```

(12) **F2.** Defina uma versão eficiente de *flatten* usando uma função auxiliar *flatten'*.

Dica: Use indução.

DEFINIÇÃO.

(4) **F3.** Enuncie uma equação interessante sobre *flatten* e *mirror*.

RESPOSTA.

```
Sum (flatten t) = Sum $ flatten (mirror t)
```

nah..

(16) **F4.** Enuncie e demonstre uma equação interessante sobre *ntips* e *nforks*.
RESPOSTA.

```
ntips t = (nforks t) + 1
type error
```

(10) **F5.** Desenhe os diagramas comutativos que correspondem aos teoremas **F3** e **F4**.
RESPOSTA.

(8) **F6.** Modifique o Tree para representar arvores que carregam informações não apenas nas folhas, mas nos nós também, mantendo a natureza binária.
RESPOSTA.

```
data Tree α where
  empty :: Tree α
  fork  :: α → Tree β → Tree γ → Tree α
```

"hi" 2 3 !

? ?

X

(8) **F7.** Qual seria um tipo razoável para a *flatten* desse tipo de arvores?
Defina tal *flatten* (não se preocupe com eficiência).
RESPOSTA.

```
flatten :: Tree α → List α
flatten empty = []
flatten (fork a l r) = (a : flatten l) ++ flatten r
```

X

F

Para qualquer α : Type, definimos

```
data Tree a                                  data Dir        type Path = List Dir
  Tip  : a → Tree a                            L : Dir
  Fork : Tree a → Tree a → Tree a              R : Dir
```

(12) **F1.** Defina as funções:

```
depth   : Tree a → Nat          flatten : Tree a → List a
ntips   : Tree a → Nat          mirror  : Tree a → Tree a
nforks  : Tree a → Nat          fetch   : Tree a → Path → Maybe a
```

DEFINIÇÕES.

```
depth (Tips _) = 0
depth (Fork l r) = S (max (depth l, depth r))

ntips (Tips _) = S0
ntips (Fork l r) = ntips l + ntips r

nforks (Tips _) = 0
nforks (Fork l r) = S (nforks l + nforks r)

flatten (Tip x) = [x]
flatten (Fork l r) = flatten l ++ flatten r

mirror (Tip x) = Tip x
mirror (Fork l r) = Fork (mirror r) (mirror l)

fetch (Tip x) [] = Just x
fetch (Fork l r) (L::ds) = fetch l ds
fetch (Fork l r) (R::ds) = fetch r ds
fetch —  —  = Nothing
```

(12) **F2.** Defina uma versão eficiente de *flatten* usando uma função auxiliar *flatten'*.

Dica: Use indução.

DEFINIÇÃO.

```
flatten t = flatten' t []

flatten' : Tree α → List α → List α
flatten' (Tip x) xs = x::xs
flatten' (Fork l r) xs = flatten' l (flatten' r xs)
```

(4) **F3.** Enuncie uma equação interessante sobre *flatten* e *mirror*.

RESPOSTA.

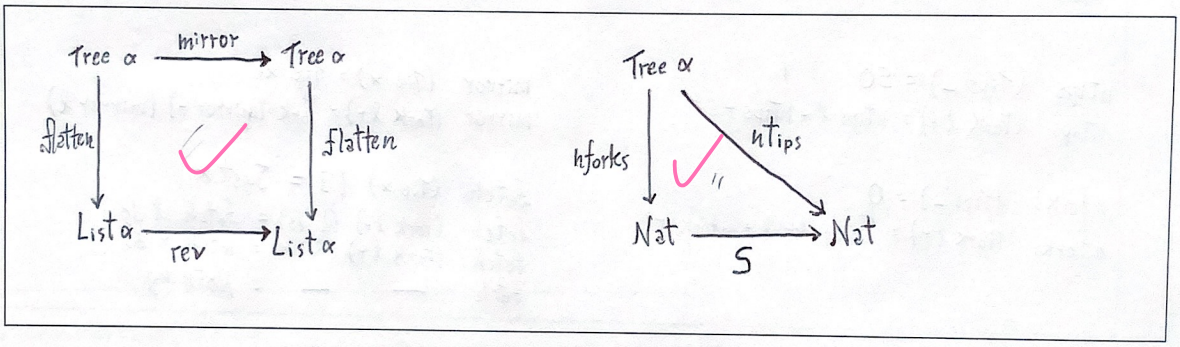
```
flatten ∘ mirror = rev ∘ flatten
```

(16) **F4.** Enuncie e demonstre uma equação interessante sobre *ntips* e *nforks*.
RESPOSTA.

Θ. (∀t: Tree α) [ntips t = S(nforks t)]
DEMONS:
Indução.
BASE:
~~"t = Tip x"~~ Seja x : α.
Calculamos:
ntips (Tip x) = S0. [ntips.1]
S(nforks (Tip x)) = S 0. [nforks.1]

PASSO INDUTIVO:
Sejam l, r : Tree α t.q. ntips l = S(nforks l) (HL)
& ntips r = S(nforks r). (HR)
Calculamos:
ntips (Fork l r) = ntips l + ntips r [ntips.2]
= S(nforks l) + ntips r [HL]
= S(nforks l) + S(nforks r) [HR]
= SS (nforks l + nforks r). [(+).2; succ.2]
S(nforks (Fork l r)) = S(S(nforks l + nforks r)) [nforks2]

(10) **F5.** Desenhe os diagramas comutativos que correspondem aos teoremas **F3** e **F4**.
RESPOSTA.



(8) **F6.** Modifique o Tree para representar arvores que carregam informações não apenas nas folhas, mas nos nós também, mantendo a natureza binária.
RESPOSTA.

```
data Tree α β
  Tip : α → Tree α β
  Fork : β → Tree α β → Tree α β → Tree α β
```

(8) **F7.** Qual seria um tipo razoável para a *flatten* desse tipo de arvores?
Defina tal *flatten* (não se preocupe com eficiência).
RESPOSTA.

```
flatten : Tree α β → List (α + β)
flatten (Tip a) = [l. a]
flatten (Fork b l r) = flatten l ++ [r. b] ++ flatten r
```

3) **E**

Considere o tipo de expressões aritméticas envolvendo apenas números e divisões inteiras:

```
data Expr
  Val : Int → Expr
  Div : Expr → Expr → Expr
```

(4) **E1.** Dada a função *div* que retorna o quociente explique qual o problema da função seguinte:

```
eval : Expr → Int
eval (Val n) = n
eval (Div u v) = div (eval u) (eval v)
```

RESPOSTA.

Se, em algum momento "(eval v)" (o 2º argumento) = 0, então não há resposta para isso nos inteiros, e pode ser considerado um erro na própria div ou um caso de não-terminação do programa.

(8) **E2.** Defina uma versão segura (total) de *div* (chame de *safediv*).
DEFINIÇÃO.

```
safediv : Int × Int → Maybe Int
safediv (_, 0) = Nothing
safediv (n,m)  | n ≥ m = Just (div (n-m)(m) + 1)
               | otherwise = Just 0
```

?? tá tentando re-implementar a div?

(24) **E3.** Usando tua função *safediv* defina uma melhor *eval* : Expr → ?.
DEFINIÇÃO.

```
eval : Expr → Maybe Int
eval (Val n) = Just n
eval (Div u v) = stripexMaybe ( pure safediv <*> eval u <*> eval v)

stripexMaybe : Maybe (Maybe α) → Maybe α
stripexMaybe (Just (Just x)) = Just x
stripexMaybe _               = Nothing
```

(12) **E4.** Um aluno—talvez não o melhor da turma—respondeu na **E3** assim:

```
eval : Expr → Maybe Int
eval (Val n)   = pure n
eval (Div u v) = pure safediv <*> eval u <*> eval v
                 α→Aα          M Int      M Int
```

Compila? Se não, explique o porquê. Se sim, explique porque seria melhor evitar esta definição.
RESPOSTA.

Não. Como o tipo de safediv : Int → Int → Maybe Int, considerando associatividade-L sintática do <*>; M (α→β) → Mα → Mβ, temos, no final, algo do tipo M (M Int).

Só isso mesmo.

J) **F**

Para qualquer α : Type, definimos

```
data Tree a                              data Dir        type Path = List Dir
  Tip  : a → Tree a                        L : Dir
  Fork : Tree a → Tree a → Tree a          R : Dir
```

(12) **F1.** Defina as funções:

```
depth   : Tree a → Nat          flatten : Tree a → List a
ntips   : Tree a → Nat          mirror  : Tree a → Tree a
nforks  : Tree a → Nat          fetch   : Tree a → Path → Maybe a
```

DEFINIÇÕES.

```
depth (Tip _) = 0
depth (Fork l r) = max (depth l) (depth r) + 1

ntips (Tip _) = 1
ntips (Fork l r) = ntips l + ntips r

nforks (Tip _) = 0
nforks (Fork l r) = nforks l + nforks r + 1
```

```
flatten (Tip x) = [x]
flatten (Fork l r) = flatten l ++ flatten r

mirror (Fork l r) = Fork (mirror r) (mirror l)
mirror x = x

fetch [] (Tip x) = Just x
fetch ds (Tip x) = Nothing
fetch [] (Fork _ _) = Nothing
fetch (L:ds) (Fork l _) = fetch ds l
fetch (R:ds) (Fork _ r) = fetch ds r
```

(12) **F2.** Defina uma versão eficiente de *flatten* usando uma função auxiliar *flatten'*.

Dica: Use indução.

DEFINIÇÃO.

Vish! Não desenrolei :(

(4) **F3.** Enuncie uma equação interessante sobre *flatten* e *mirror*.

RESPOSTA.

```
reverse ∘ flatten = flatten ∘ mirror
```

flatmirror = flatten ∘ mirror

flatten = reverse ∘ flatmirror

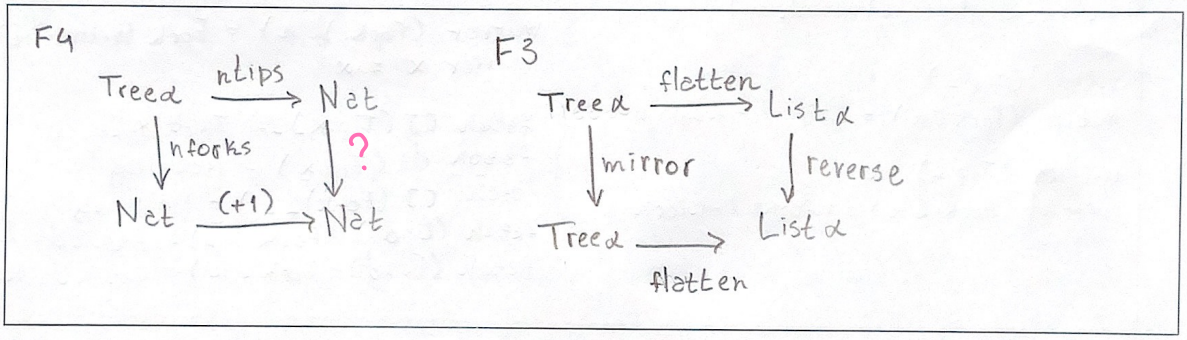
(16) **F4.** Enuncie e demonstre uma equação interessante sobre *ntips* e *nforks*.
RESPOSTA.

ntips t = nforks t + 1
Por indução no t.
Caso Tip x:
ntips (Tip x)
= 1
nforks (Tip x) + 1
= 0 + 1
= 1 ✓

Caso Fork l r:
Suponha que ntips l = nforsk l + 1 e
a mesma coisa pra r. (hi)
Calc:
ntips (Fork l r)
= ntips l + ntips r
= nforks l + 1 + nforks r + 1 (hi)
nforks (Fork l r) + 1
= (nforks l + nforks r) + 1 + 1
donde chegou?

Pela Ass. e Com., se tornam iguais. ← yikes!

(10) **F5.** Desenhe os diagramas comutativos que correspondem aos teoremas **F3** e **F4**.
RESPOSTA.



(8) **F6.** Modifique o Tree para representar arvores que carregam informações não apenas nas folhas, mas nos nós também, mantendo a natureza binária.
RESPOSTA.

```
data Tree α β where
  Node :: α → Tree α β → Tree α β → Tree α β
  Leaf :: β → Tree α β
```
✓

(8) **F7.** Qual seria um tipo razoável para a *flatten* desse tipo de arvores?
Defina tal *flatten* (não se preocupe com eficiência).
RESPOSTA.

```
flatten :: Tree α → List (Either α β)
flatten (Leaf x) = [Right x]
flatten (Node x l r) = Left x : (flatten l ++ flatten r)
```

(28) **E**

Considere o tipo de expressões aritméticas envolvendo apenas números e divisões inteiras:

```
data Expr
  Val : Int → Expr
  Div : Expr → Expr → Expr
```

(4) **E1.** Dada a função *div* que retorna o quociente explique qual o problema da função seguinte:

```
eval : Expr → Int
eval (Val n) = n
eval (Div u v) = div (eval u) (eval v)
```

RESPOSTA.

A div não é total e pode retornar erro. Quando??

(8) **E2.** Defina uma versão segura (total) de *div* (chame de *safediv*).
DEFINIÇÃO.

```
safediv _ 0 = Nothing
safediv x y = Just $ div x y
```

safediv :: Int → Int → Maybe Int

(24) **E3.** Usando tua função *safediv* defina uma melhor *eval* : Expr → ?.
DEFINIÇÃO.

```
eval :: Expr → Maybe Int
eval (Val n) = Just n
eval (Div u v) = case eval u of
                   Just x → safediv x <$> eval v
                   Nothing → Nothing
```

TYPE ERROR:
deveria ser: Int → Int
mas é: Int → M Int

(12) **E4.** Um aluno—talvez não o melhor da turma—respondeu na **E3** assim:

```
eval : Expr → Maybe Int
eval (Val n)   = pure n
eval (Div u v) = pure safediv <*> eval u <*> eval v
```

Compila? Se não, explique o porquê. Se sim, explique porque seria melhor evitar esta definição.
RESPOSTA.

Não! pure safediv tem tipo 'Maybe (Int → Int → Maybe Int)'. Enquanto '<*>' esperava 'Maybe (Expr → Int)'.

Int → Int → Int

Só isso mesmo.

(10) **F**

Para qualquer α : Type, definimos

```
data Tree a                                  data Dir        type Path = List Dir
  Tip  : a → Tree a                            L : Dir
  Fork : Tree a → Tree a → Tree a              R : Dir
```

(12) **F1.** Defina as funções:

```
depth   : Tree a → Nat          flatten : Tree a → List a
ntips   : Tree a → Nat          mirror  : Tree a → Tree a
nforks  : Tree a → Nat          fetch   : Tree a → Path → Maybe a
```

DEFINIÇÕES.

```
depth (Tip a) = 0
depth (Fork t u) = S (max (depth t) (depth u))

ntips (Tip a) = S0
ntips (Fork t u) = ntips t + ntips u

nforks (Tip a) = 0
nforks (Fork t u) = S (nforks t + nforks u)
```

```
flatten (Tip a) = [a]
flatten (Fork t u) =
flatten t ++ flatten u

mirror (Tip a) = Tip a
mirror (Fork t u) = ? u não é tão vizinho
Fork (mirror u) (mirror t)

fetch (Tip a) [] = Just a
fetch (Tip a) _ = Nothing
fetch (Fork t u) [] = Nothing
fetch (Fork t u) (d::ds) =
match d:
  L = fetch t ds
  R = fetch u ds
```

(12) **F2.** Defina uma versão eficiente de *flatten* usando uma função auxiliar *flatten'*.

Dica: Use indução.

DEFINIÇÃO.

(4) **F3.** Enuncie uma equação interessante sobre *flatten* e *mirror*.

RESPOSTA.

```
flatten t = reverse (flatten (mirror t))
```

Simplify!

(16) **F4.** Enuncie e demonstre uma equação interessante sobre *ntips* e *nforks*.
RESPOSTA.

ntips t = S (nforks t)
Por indução no t.
CASO (Tip a):
ntips (Tip a) = S 0 [ntips.1]
S (nforks (Tip a)) = S (0) [nforks.1]
= S 0 ?!
CASO (Fork u v):
ntips (Fork u v) = ntips u + ntips v
nforks (Fork u v) = S (nforks u + nforks v) ...?

Como u e v precisam a ... ntips

(10) **F5.** Desenhe os diagramas comutativos que correspondem aos teoremas **F3** e **F4**.
RESPOSTA.

(8) **F6.** Modifique o Tree para representar arvores que carregam informações não apenas nas folhas, mas nos nós também, mantendo a natureza binária.
RESPOSTA.

```
data Tree a b
  Tip : a → Tree a b
  Fork : b → Tree a b → Tree a b → Tree a b
```

(8) **F7.** Qual seria um tipo razoável para a *flatten* desse tipo de arvores?
Defina tal *flatten* (não se preocupe com eficiência).
RESPOSTA.

```
flatten : Tree a b → List (Either a b)
flatten (Tip a) = [Either a]
flatten (b (Fork u v)) = flatten u ++ [Either b] ++ flatten v
```

quais são os construtores de Either α β?

(70) **F**

Para qualquer α : Type, definimos

```
data Tree a                                  data Dir        type Path = List Dir
  Tip  : a → Tree a                            L : Dir
  Fork : Tree a → Tree a → Tree a              R : Dir
```

(12) **F1.** Defina as funções:

```
depth   : Tree a → Nat          flatten : Tree a → List a
ntips   : Tree a → Nat          mirror  : Tree a → Tree a
nforks  : Tree a → Nat          fetch   : Tree a → Path → Maybe a
```

DEFINIÇÕES.

```
nforks Tip = 0
nforks
```

(12) **F2.** Defina uma versão eficiente de *flatten* usando uma função auxiliar *flatten'*.

Dica: Use indução.

DEFINIÇÃO.

(4) **F3.** Enuncie uma equação interessante sobre *flatten* e *mirror*.

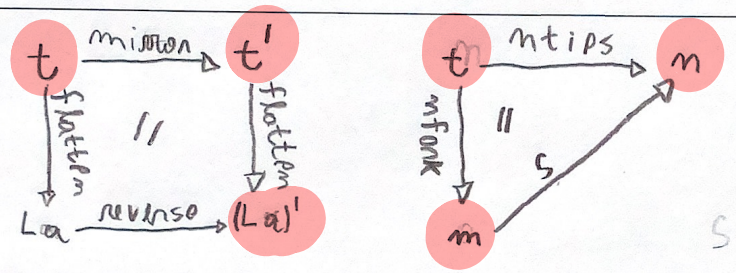
RESPOSTA.

```
flatten (mirror t) = reverse (flatten t)
```

(16) **F4.** Enuncie e demonstre uma equação interessante sobre *ntips* e *nforks*.
RESPOSTA.

ntips t = S (nforks t)

(10) **F5.** Desenhe os diagramas comutativos que correspondem aos teoremas **F3** e **F4**.
RESPOSTA.



deveriam ser tipos!

(8) **F6.** Modifique o Tree para representar arvores que carregam informações não apenas nas folhas, mas nos nós também, mantendo a natureza binária.
RESPOSTA.

(8) **F7.** Qual seria um tipo razoável para a *flatten* desse tipo de arvores?
Defina tal *flatten* (não se preocupe com eficiência).
RESPOSTA.

(48) **E**

Considere o tipo de expressões aritméticas envolvendo apenas números e divisões inteiras:

```
data Expr
  Val : Int → Expr
  Div : Expr → Expr → Expr
```

(4) **E1.** Dada a função *div* que retorna o quociente explique qual o problema da função seguinte:

```
eval : Expr → Int
eval (Val n) = n
eval (Div u v) = div (eval u) (eval v)
```

RESPOSTA.

A função não trata a eventualidade de uma divisão por 0, sempre prometendo entregar um Int. Logo, erro.

(8) **E2.** Defina uma versão segura (total) de *div* (chame de *safediv*).
DEFINIÇÃO.

```
safediv : Maybe Int → Maybe Int → Maybe Int
safediv _ (Just 0) = Nothing
safediv _ Nothing = Nothing
safediv (Just n) (Just m) = Just (div n m)
```

(24) **E3.** Usando tua função *safediv* defina uma melhor *eval* : Expr → ?.
DEFINIÇÃO.

```
eval : Expr → Maybe Int
eval (Val n) = Just n
eval (Div u v) = safediv (eval u) (eval v)
```

não sendo honesta a safediv meio que robou aqui!

(12) **E4.** Um aluno—talvez não o melhor da turma—respondeu na **E3** assim:

```
eval : Expr → Maybe Int
eval (Val n)   = pure n
eval (Div u v) = pure safediv <*> eval u <*> eval v
```

Compila? Se não, explique o porquê. Se sim, explique porque seria melhor evitar esta definição.
RESPOSTA.

Acho que sim. Por causa da forma que foi construída, essa definição exige mais esforço ao ser utilizada, isto é, não é eficiente.

X

Só isso mesmo.

(70) **F**

Para qualquer α : Type, definimos

```
data Tree a                                  data Dir        type Path = List Dir
  Tip  : a → Tree a                            L : Dir
  Fork : Tree a → Tree a → Tree a              R : Dir
```

(12) **F1.** Defina as funções:

```
depth   : Tree a → Nat          flatten : Tree a → List a
ntips   : Tree a → Nat          mirror  : Tree a → Tree a
nforks  : Tree a → Nat          fetch   : Tree a → Path → Maybe a
```

DEFINIÇÕES.

```
depth (Tip x) = 0
depth (Fork at bt) = max (S (depth at)) (S (depth bt))

ntips (Tip _) = S 0
ntips (Fork at bt) = ntips at + ntips bt

nforks (Tip _) = 0
nforks (Fork at bt) = S (nforks at + nforks bt)

fetch (Tip x) [] = Just x
fetch (Fork at bt) (d : ds) = case d of L -> fetch at ds
                                        R -> fetch bt ds
fetch _ _ = Nothing

flatten (Tip x) = [x]
flatten (Fork at bt) =
  flatten at ++ flatten bt

mirror (Tip x) = Tip x
mirror (Fork at bt) = Fork bt at
```

(12) **F2.** Defina uma versão eficiente de *flatten* usando uma função auxiliar *flatten'*.

Dica: Use indução.

DEFINIÇÃO.

```
flatten' ::
```

(4) **F3.** Enuncie uma equação interessante sobre *flatten* e *mirror*.

RESPOSTA.

```
flatten . mirror = reverse . flatten
```

(+).com = comutatividade

(16) **F4.** Enuncie e demonstre uma equação interessante sobre *ntips* e *nforks*.
RESPOSTA.

(∀ at :: Tree α) [ntips at = S (nforks at)] ✓
Seja t :: Tree α.

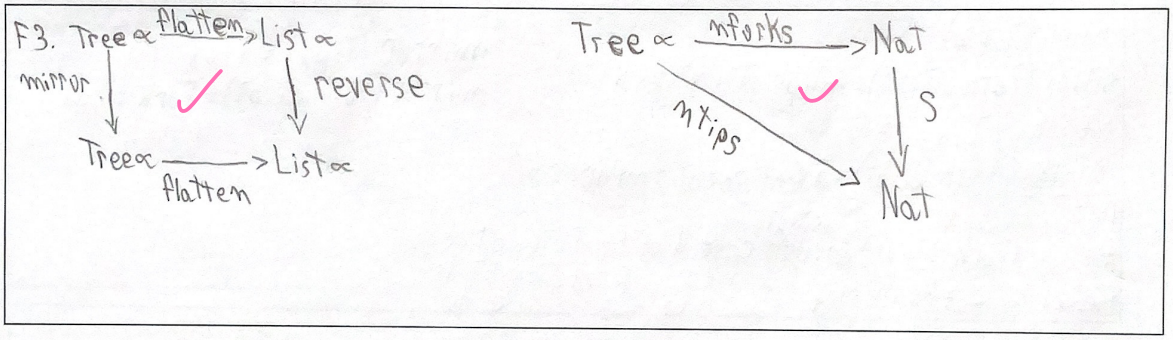
Caso Tip x:
ntips (Tip x)
= [ntips.1]
S 0
S (nforks (Tip x))
= [nforks.1]
S 0 ✓

Caso Fork at bt:
ntips (Fork at bt)
= [ntips.2 at := at bt := bt]
ntips at + ntips bt
= [H.I.]
S (nforks at) + S (nforks bt)
= [(+).2 n m]
S (S (nforks at) + nforks bt)

(+) :: Nat → Nat → Nat
n + 0 = n
n + S m = S (n + m)

= [(+).com e (+).2]
S (S (nforks at + nforks bt))
= [nforks.2 at := at bt := bt]
S (nforks (Fork at bt)) ✓

(10) **F5.** Desenhe os diagramas comutativos que correspondem aos teoremas **F3** e **F4**.
RESPOSTA.



(8) **F6.** Modifique o Tree para representar arvores que carregam informações não apenas nas folhas, mas nos nós também, mantendo a natureza binária.
RESPOSTA.

Nós de diferentes tipos

quem é α?

```
data Node α
  Node :: α → Node α

data Tree
  Tip :: Node α → Tree
  Parent :: Node α → Tree → Tree → Tree
```

qual é o propósito desse tipo?

"hi"
3 5

(8) **F7.** Qual seria um tipo razoável para a *flatten* desse tipo de arvores?
Defina tal *flatten* (não se preocupe com eficiência). EM ORDEM
RESPOSTA.

f

(48) **E**

Considere o tipo de expressões aritméticas envolvendo apenas números e divisões inteiras:

```
data Expr
  Val : Int → Expr
  Div : Expr → Expr → Expr
```

(4) **E1.** Dada a função *div* que retorna o quociente explique qual o problema da função seguinte:

(4/9)/42

```
eval : Expr → Int
eval (Val n) = n
eval (Div u v) = div (eval u) (eval v)
```

RESPOSTA.

Daria erro numa divisão por zero.

(8) **E2.** Defina uma versão segura (total) de *div* (chame de *safediv*).
DEFINIÇÃO.

```
Safediv :: Int -> Int -> Maybe Int
Safediv _ 0 = Nothing
Safediv m m = Just (div m m)
```

(24) **E3.** Usando tua função *safediv* defina uma melhor *eval* : Expr → ?.
DEFINIÇÃO.

(12) **E4.** Um aluno—talvez não o melhor da turma—respondeu na **E3** assim:

```
eval : Expr → Maybe Int
eval (Val n)   = pure n → Just n
eval (Div u v) = pure safediv <*> eval u <*> eval v
```

Compila? Se não, explique o porquê. Se sim, explique porque seria melhor evitar esta definição.
RESPOSTA.

Só isso mesmo.

F

Para qualquer α : Type, definimos

```
data Tree a                                  data Dir        type Path = List Dir
  Tip  : a → Tree a                            L : Dir
  Fork : Tree a → Tree a → Tree a              R : Dir
```

(12) **F1.** Defina as funções:

```
depth   : Tree a → Nat          flatten : Tree a → List a
ntips   : Tree a → Nat          mirror  : Tree a → Tree a
nforks  : Tree a → Nat          fetch   : Tree a → Path → Maybe a
```

DEFINIÇÕES.

```
depth (Fork t s) = S (max (depth t) (depth s))
depth (Tip _)    = 0

ntips (Fork t s) = ntips t + ntips s
ntips (Tip _)    = 1

nforks (Fork t s) = 1 + nforks t + nforks s
nforks (Tip _)    = 0

flatten (Fork t s) = flatten t ++ flatten s
flatten (Tip a)    = [a]

mirror (Tip a)    = Tip a
mirror (Fork t s) = Fork (mirror s) (mirror t)

fetch (L::_) (Fork (Tip a) _) = Just a
fetch (R::_) (Fork _ (Tip a)) = Just a
fetch (L::p) (Fork t s)       = fetch p t
fetch (R::p) (Fork t s)       = fetch p t
fetch _ _                     = Nothing
```

x está na posição [] no Tip x.

(12) **F2.** Defina uma versão eficiente de *flatten* usando uma função auxiliar *flatten'*.

Dica: Use indução.

DEFINIÇÃO.

(4) **F3.** Enuncie uma equação interessante sobre *flatten* e *mirror*.

RESPOSTA.

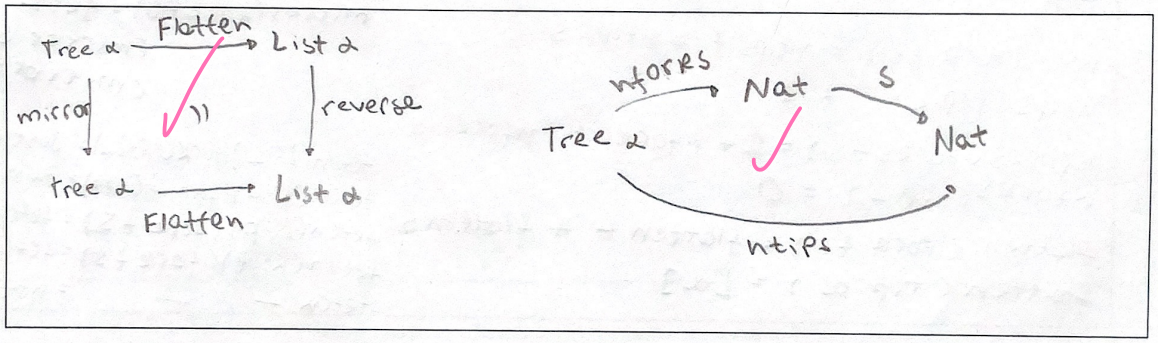
```
flatten ∘ mirror = reverse ∘ flatten
```

(16) **F4.** Enuncie e demonstre uma equação interessante sobre *ntips* e *nforks*.
RESPOSTA.

```
ntips = S ∘ nforks

seja t : Tree α
caso t = tip a
  cor: ntips (tip a) = 1 [ntips.1]
  cor: (S ∘ nforks) (tip a) = S (nforks (tip a)) [∘.1]
                            = S 0 [nforks.2]
caso t = Fork s t
  cor: ntips (Fork s t) = ntips s + ntips t [ntips.1]
                        = (S ∘ nforks) s + (S ∘ nforks) t [HI.]
                        = 1 + nforks s + 1 + nforks t [∘.1; (+1)←]
                        = 1 + (1 + nforks s + nforks t) [+.assoc; +.com]
                        = 1 + nforks s t [nforks.2]
```

(10) **F5.** Desenhe os diagramas comutativos que correspondem aos teoremas **F3** e **F4**.
RESPOSTA.



(8) **F6.** Modifique o Tree para representar arvores que carregam informações não apenas nas folhas, mas nos nós também, mantendo a natureza binária.
RESPOSTA.

```
data Tree α β
   tip : α → Tree α β
   Fork : β → Tree α β → Tree α β → Tree α β
```

(8) **F7.** Qual seria um tipo razoável para a *flatten* desse tipo de arvores?
Defina tal *flatten* (não se preocupe com eficiência).
RESPOSTA.

```
flatten : Tree α β → List α+β
flatten (Fork b t s) = flatten t # [Right b] # flatten s
flatten (tip a)      = [Left a]
```

8) **E**

Considere o tipo de expressões aritméticas envolvendo apenas números e divisões inteiras:

```
data Expr
  Val : Int → Expr
  Div : Expr → Expr → Expr
```

(4) **E1.** Dada a função *div* que retorna o quociente explique qual o problema da função seguinte:

```
eval : Expr → Int
eval (Val n) = n
eval (Div u v) = div (eval u) (eval v)
```

RESPOSTA.

A eval não termina nos casos de divisão por 0.

(8) **E2.** Defina uma versão segura (total) de *div* (chame de *safediv*).
DEFINIÇÃO.

```
Safediv : Int → Int → Maybe Int
Safediv  0  _  = Nothing
Safediv  x  y  = Just (div x y)
```

(24) **E3.** Usando tua função *safediv* defina uma melhor *eval* : Expr → ?.
DEFINIÇÃO.

```
eval : Expr → Maybe Int
eval (Val n) = Just n
eval (Div u v) = case (safediv u, safediv v) of
                   (Just u, Just v) → Safediv u v
                   (_ , _) → Nothing
```

poderia ser apenas _

(12) **E4.** Um aluno—talvez não o melhor da turma—respondeu na **E3** assim:

```
eval : Expr → Maybe Int
eval (Val n)   = pure n
eval (Div u v) = pure safediv <*> eval u <*> eval v
```

Safediv : Int → Int → Int

Compila? Se não, explique o porquê. Se sim, explique porque seria melhor evitar esta definição.
RESPOSTA.

A função compila. Entretanto, é melhor evitar essa definição por questões de eficiência. todo o (eval u) da segunda equação seria evaluado antes de olhar o (eval v), que poderia ser Nothing.

X

Só isso mesmo.

(/0) **F**

Para qualquer α : Type, definimos

```
data Tree a                                   data Dir        type Path = List Dir
  Tip  : a → Tree a                             L : Dir
  Fork : Tree a → Tree a → Tree a               R : Dir
```

(12) **F1.** Defina as funções:

```
depth   : Tree a → Nat          flatten : Tree a → List a
ntips   : Tree a → Nat          mirror  : Tree a → Tree a
nforks  : Tree a → Nat          fetch   : Tree a → Path → Maybe a
```

DEFINIÇÕES.

```
depth : Tree α -> Nat
depth (tip _) = 0
depth (fork l r) = S(max(depth l, depth r))

ntips : Tree α -> Nat
ntips (Tip _) = S0
ntips (fork l r) = ntips l + ntips r

nforks : Tree α -> Nat
nforks (Tip _) = 0
nforks (fork l r) = S(nforks l + nforks r)

flatten : Tree α -> List α
flatten (Tip n) = [n]
flatten (fork l r) = flatten l ++ flatten r

mirror : Tree α -> Tree α
mirror (Tip n) = Tip n
mirror (fork l r) = fork (mirror r) (mirror l)

fetch : Tree α -> Path -> Maybe α
fetch (Tip n) [] = Just n
fetch (fork l r) (d::ds) = case d of
  L -> fetch l ds
  R -> fetch r ds
fetch _ _ = Nothing
```

(12) **F2.** Defina uma versão eficiente de *flatten* usando uma função auxiliar *flatten'*.

Dica: ~~Use indução.~~

DEFINIÇÃO.

Sacada da revcat que eu esqueci

```
flatten : Tree α -> List α
flatten (Tip n) = [n]
flatten (fork l r) = flatten' (flatten l) (flatten r)

flatten' : List α -> List α -> List α
```
Tree

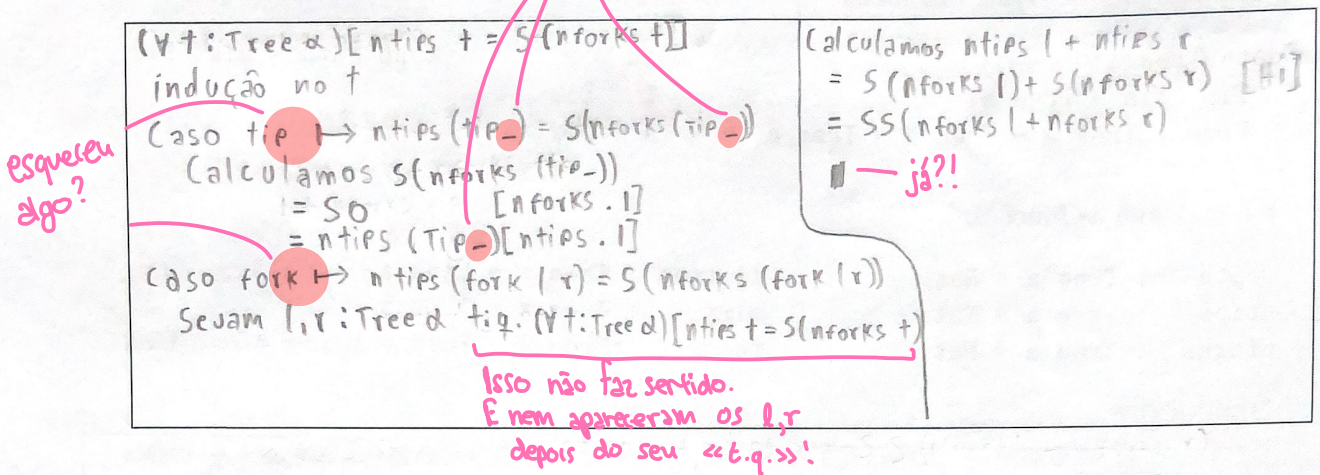
(4) **F3.** Enuncie uma equação interessante sobre *flatten* e *mirror*.

RESPOSTA.

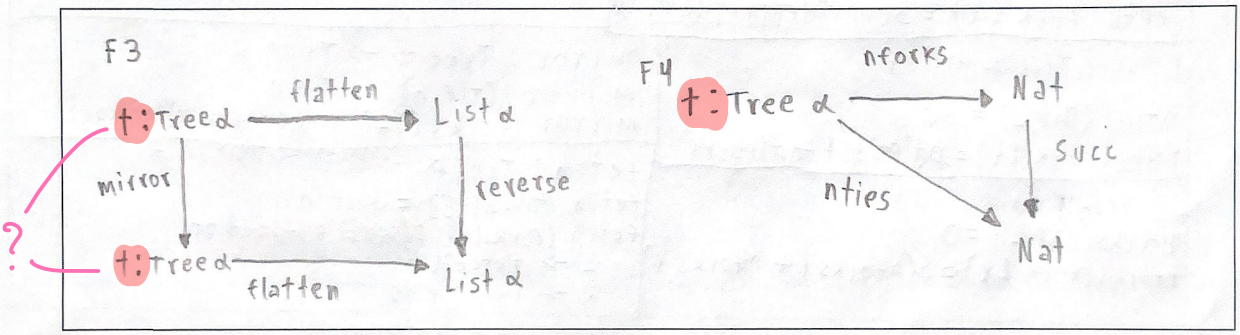
```
reverse ∘ flatten = flatten ∘ mirror
```

parece ter mal-entendido sobre o − !

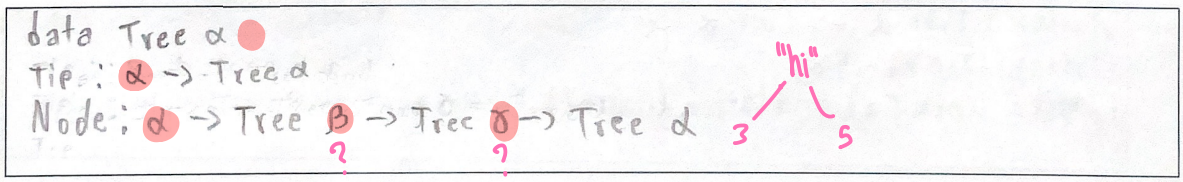
(16) **F4.** Enuncie e demonstre uma equação interessante sobre *ntips* e *nforks*.
RESPOSTA.



(10) **F5.** Desenhe os diagramas comutativos que correspondem aos teoremas **F3** e **F4**.
RESPOSTA.



(8) **F6.** Modifique o Tree para representar arvores que carregam informações não apenas nas folhas, mas nos nós também, mantendo a natureza binária.
RESPOSTA.



(8) **F7.** Qual seria um tipo razoável para a *flatten* desse tipo de arvores?
Defina tal *flatten* (não se preocupe com eficiência).
RESPOSTA.

(48) **E**

Considere o tipo de expressões aritméticas envolvendo apenas números e divisões inteiras:

```
data Expr
  Val : Int → Expr
  Div : Expr → Expr → Expr
```

(4) **E1.** Dada a função *div* que retorna o quociente explique qual o problema da função seguinte:

```
eval : Expr → Int
eval (Val n) = n
eval (Div u v) = div (eval u) (eval v)
```

RESPOSTA.

✓ div é uma função parcial, que retorna um erro caso v=0

(8) **E2.** Defina uma versão segura (total) de *div* (chame de *safediv*).
DEFINIÇÃO.

```
Safediv : Int → Int → Maybe Int
Safediv _ 0 = Nothing
Safediv n m = Just (div n m)
```

✓

(24) **E3.** Usando tua função *safediv* defina uma melhor *eval* : Expr → ?.
DEFINIÇÃO.

```
eval : Expr -> Maybe Int
eval (Val n) = Just n
eval (Div u v) = safediv n m
```

quem é?

type error!

(12) **E4.** Um aluno—talvez não o melhor da turma—respondeu na **E3** assim:

```
eval : Expr → Maybe Int
eval (Val n)   = pure n
eval (Div u v) = pure (safediv <*> eval u) <*> eval v)
```

as parens não poderiam ser assim! (Por quê?)

Compila? Se não, explique o porquê. Se sim, explique porque seria melhor evitar esta definição.
RESPOSTA.

Compila, mas adiciona uma camada de complexidade desnecessária na resolução

Só isso mesmo.

X

(*) : F(α→β) → F α → F β

Pure : α → F α

(22) **G**

(10) **G1.** Complete as equações seguintes **com algo interessante:**[3]

X `map f (flatten t)` = MAP F (REV (FLATTEN (MIRROR T)))

✓ `sum (replicate n x)` = n.X

✓ `sum (map (+ k) ns)` = ~~Sum (NS) +~~ k.(LEN NS) + Sum NS

✓ `sum (map (· k) ns)` = k.(Sum ns)

~ `sorted (map (+ k) ns)` = map (+k) (Bool (SORTED) NS)

confundiu sort : L α → L α

Sorted . L α → B

type error

(12) **G2.** Escolha **uma** das equações de **G1** para demonstrar. Precisas definir (corretamente!) todas as funções envolvidas, exceto se foram definidas em outras questões desta prova.

DEFINIÇÕES.

```
Sum : List Nat → Nat
Sum [] = 0
Sum (x:xs) = x + Sum xs
```

✓

```
Replicate : Nat → Nat → List Nat
Replicate 0 _ = []
Replicate Sn m = m:(Replicate n m)
```

✓

DEMONSTRAÇÃO DE Sum (REPLICATE n x) = n·x .

Por indução no n

Caso n = 0:

Sum (REPLICATE 0 x)
= Sum [] [REPLICATE.1]
= 0 [SUM.1]

0 x
= 0

✓

CASO n = Sk:

~~Sum (REPLICATE (k+Sn) x)~~
~~= Sum (REPLICATE S(k+n) x)~~

Sum (REPLICATE Sk x)
= Sum (x : (REPLICATE k x)) [REPLICATE.2]
= x + SUM (REPLICATE k x) [SUM.2]
= x + kx [H.I.]
= Sk.x

Sn = Sk
m = x
x = x
xs = REPLICATE k x

✓

[3] DEFINIÇÃO. Chamamos algo de *interessante* sse Thanos acha tal algo interessante.

(70) **F**

Para qualquer α : Type, definimos

```
data Tree a
  Tip  : a → Tree a
  Fork : Tree a → Tree a → Tree a

data Dir
  L : Dir
  R : Dir

type Path = List Dir
```

(12) **F1.** Defina as funções:

```
depth   : Tree a → Nat
ntips   : Tree a → Nat
nforks  : Tree a → Nat

flatten : Tree a → List a
mirror  : Tree a → Tree a
fetch   : Tree a → Path → Maybe a
```

DEFINIÇÕES.

(12) **F2.** Defina uma versão eficiente de *flatten* usando uma função auxiliar *flatten′*.

Dica: Use indução.

DEFINIÇÃO.

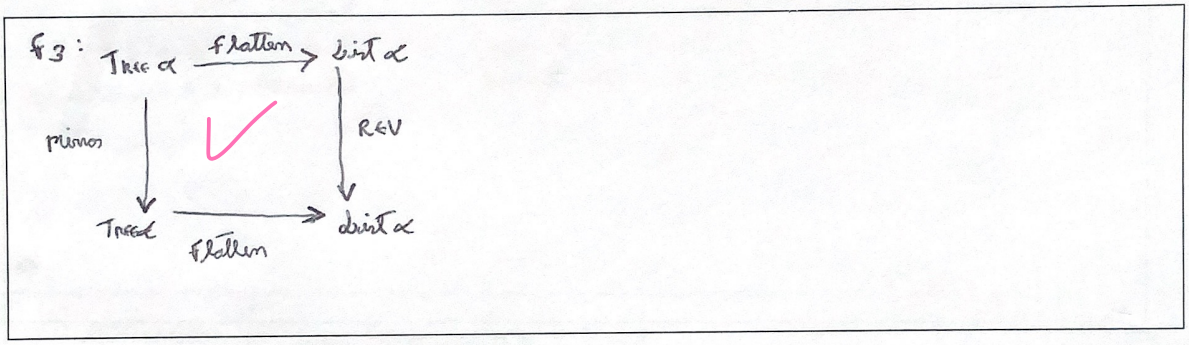
(4) **F3.** Enuncie uma equação interessante sobre *flatten* e *mirror*.

RESPOSTA.

flatten (mirror T) = REV (Flatten T)

(16) **F4.** Enuncie e demonstre uma equação interessante sobre *ntips* e *nforks*.
RESPOSTA.

(10) **F5.** Desenhe os diagramas comutativos que correspondem aos teoremas **F3** e **F4**.
RESPOSTA.



(8) **F6.** Modifique o Tree para representar arvores que carregam informações não apenas nas folhas, mas nos nós também, mantendo a natureza binária.
RESPOSTA.

(8) **F7.** Qual seria um tipo razoável para a *flatten* desse tipo de arvores?
Defina tal *flatten* (não se preocupe com eficiência).
RESPOSTA.

(70) **F**

Para qualquer α : Type, definimos

```
data Tree a
  Tip  : a → Tree a
  Fork : Tree a → Tree a → Tree a

data Dir
  L : Dir
  R : Dir

type Path = List Dir
```

igual

(12) **F1.** Defina as funções:

```
depth   : Tree a → Nat
ntips   : Tree a → Nat
nforks  : Tree a → Nat
flatten : Tree a → List a
mirror  : Tree a → Tree a
fetch   : Tree a → Path → Maybe a
```

DEFINIÇÕES.

```
depth (Tip _) = 0
depth (Fork l r) = 1 + max (depth l) (depth r)

ntips (Tip _) = 1
ntips (Fork l r) = nTips l + nTips r

nforks (Tip _) = 0
nforks (Fork l r) = 1 + nforks l + nforks r

flatten (Tip x) = [x]
flatten (Fork l r) = flatten l ++ flatten r

mirror (Tip x) = Tip x
mirror (Fork l r) = Fork (mirror l) (mirror r)

fetch (Tip x) [] = Just x
fetch (Fork l r) (d:ds) =
  match d with:
    L -> fetch l ds
    R -> fetch r ds.
fetch _ _ = Nothing
```

(12) **F2.** Defina uma versão eficiente de *flatten* usando uma função auxiliar *flatten'*.

Dica: Use indução.

DEFINIÇÃO.

```
flatten' (Fork (Tip x) (Tip y)) = [x, y]
flatten' (Fork (Tip x) t) = x : flatten t
flatten' (Fork t (Tip y)) = flatten t <: y

flatten (Tip x) = [x]
flatten t = flatten' t
```

X

demora!

:P

(4) **F3.** Enuncie uma equação interessante sobre *flatten* e *mirror*.

RESPOSTA.

```
flatten ∘ mirror = reverse ∘ flatten
```

✓

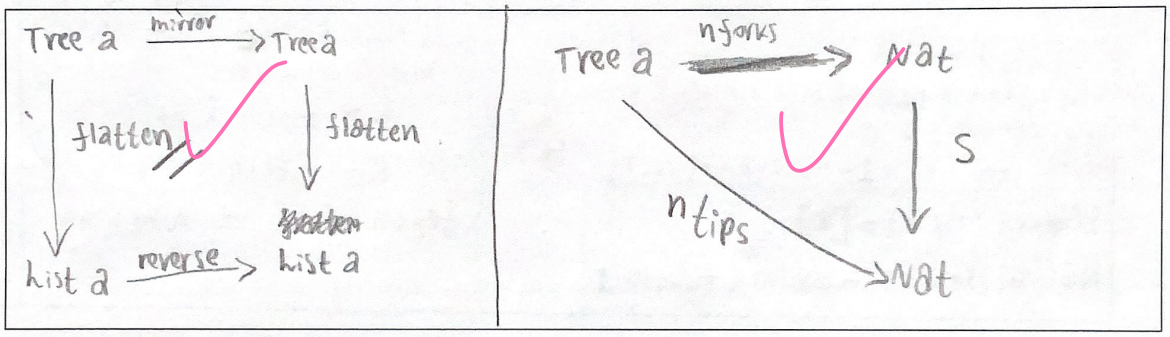
(16) **F4.** Enuncie e demonstre uma equação interessante sobre *ntips* e *nforks*.
RESPOSTA.

(S o nforks = ntips
seja t:Tree a.
caso t = (Tip x);
calculamos:
(S o nforks)(Tip x)
= S (nforks (Tip x))
= S 0
= 1.

ntips (Tip x)
= 1

caso t = (Fork l r):
calculamos:
ntips (Fork l r)
= ntips l + ntips r
= (nforks l + 1) + (nforks r + 1) [h.i.]
= (nforks l + nforks r + 1) + 1 [calcula nforks]
= (1 + nforks l + nforks r) + 1
= (nforks (Fork l r)) + 1 [nforks.2 ←]
= S (nforks (Fork l r))
= (S o nforks) t.

(10) **F5.** Desenhe os diagramas comutativos que correspondem aos teoremas **F3** e **F4**.
RESPOSTA.



(8) **F6.** Modifique o Tree para representar arvores que carregam informações não apenas nas folhas, mas nos nós também, mantendo a natureza binária.
RESPOSTA.

data Tree a b
Tip : a → Tree a b
Fork : b → Tree a b → Tree a b → Tree a b

(8) **F7.** Qual seria um tipo razoável para a *flatten* desse tipo de arvores?
Defina tal *flatten* (não se preocupe com eficiência).
RESPOSTA.

flatten : Tree a b → List (Either a b)
flatten (Tip x) = [Left x]
flatten (Fork y l r) = flatten l ++ [Right y] ++ flatten r

3/0 = Error

(48) **E**

Considere o tipo de expressões aritméticas envolvendo apenas números e divisões inteiras:

```
data Expr
  Val : Int → Expr
  Div : Expr → Expr → Expr
```

(4) **E1.** Dada a função *div* que retorna o quociente explique qual o problema da função seguinte:

```
eval : Expr → Int
eval (Val n) = n
eval (Div u v) = div (eval u) (eval v)
```

o zero

RESPOSTA.

O que ela retornaria p/ algum argumento (Div 42 0)?
Falta tratar os erros. (o problema tá na div).

(8) **E2.** Defina uma versão segura (total) de *div* (chame de *safediv*).
DEFINIÇÃO.

hm...

```
Safediv : Int → Int → Either ErrorDiv Int
Safediv _ 0 = Left (zeroDivisor "u can't divide by zero")
Safediv x y = Right (div x y)
```

```
data ErrorDiv
  ZeroDivisor : String → ErrorDiv
```

↳ qual o sentido dessa info?

(24) **E3.** Usando tua função *safediv* defina uma melhor *eval* : Expr → ?.
DEFINIÇÃO.

involutiva e [illegible] algo aí comuta.

Val é [illegible] pq eval Val n = e n = n. ?!

```
eval : Expr → Either ErrorDiv Int
eval (Val n) = Rn TypeError
eval (Div u v) = Safediv (eval u) (eval v)
```

(12) **E4.** Um aluno—talvez não o melhor da turma—respondeu na **E3** assim:

```
eval : Expr → Maybe Int
eval (Val n)   = pure n
eval (Div u v) = pure safediv <*> eval u <*> eval v
```

O APP maybe está instanciando?

Compila? Se não, explique o porquê. Se sim, explique porque seria melhor evitar esta definição.
RESPOSTA.

??

TypeError, espero que n compile porque, pelo tipo da (<*>), mas supondo que fosse MI na minha. Essa def sofre dependência na implementação do maybe como APP.
na segunda equação n retorna o tipo conforme a minha safediv,

Só isso mesmo.

M (Int → Int → Either) <*> M Int <*> M Int

<> M Either Int

(70) **F**

Para qualquer α : Type, definimos

```
data Tree a
  Tip  : a → Tree a
  Fork : Tree a → Tree a → Tree a
```

```
data Dir
  L : Dir
  R : Dir
```

```
type Path = List Dir
```

(12) **F1.** Defina as funções:

```
depth   : Tree a → Nat
ntips   : Tree a → Nat
nforks  : Tree a → Nat
flatten : Tree a → List a
mirror  : Tree a → Tree a
fetch   : Tree a → Path → Maybe a
```

DEFINIÇÕES.

```
depth (Tip _) = 0
depth (Fork l r) = S (max (depth l) (depth r))

ntips (Tip v) = S0
ntips (Fork l r) = (ntips l) + (ntips r)

nforks (Tip _) = 0
nforks (Fork l r) = S (nforks l + nforks r)

flatten (Tip v) = [v]
flatten (Fork l r) = (flatten l) ++ (flatten r)

mirror (Tip v) = Tip v
mirror (Fork l r) = Fork (mirror r) (mirror l)

fetch (Tip v) [] = Just v
fetch (Fork l _) (L :: ds) = fetch l ds
fetch (Fork _ r) (R :: ds) = fetch r ds
fetch _ _ = Nothing
```

(12) **F2.** Defina uma versão eficiente de *flatten* usando uma função auxiliar *flatten′*.

Dica: *Use indução.*

DEFINIÇÃO.

(4) **F3.** Enuncie uma equação interessante sobre *flatten* e *mirror*.

RESPOSTA.

```
(∀ t : Tree) [reverse (flatten t) = flatten (mirror t)]
```

confundiu ($) com (∘)?

(16) **F4.** Enuncie e demonstre uma equação interessante sobre *ntips* e *nforks*.
RESPOSTA.

?

$ntips^t = (+1) \circ nforks\ t$

CASO Se $t = Tip\ v$:

$ntips\ t = ntips\ (Tip\ v) = S0$ [ntips.1]

$(+1) \circ nforks\ t = (+1) \circ nforks\ (Tip\ v) = (+1)(nforks\ (Tip\ v))$ [(∘).1] $= (+1) \circ 0$ [nforks.1] $= 1 = S0$

CASO Se $t = Fork\ l\ r$

(10) **F5.** Desenhe os diagramas comutativos que correspondem aos teoremas **F3** e **F4**.
RESPOSTA.

(8) **F6.** Modifique o Tree para representar arvores que carregam informações não apenas nas folhas, mas nos nós também, mantendo a natureza binária.
RESPOSTA.

que escolha de nomes foi essa??

```
data Tree a t
  Tip : a → Tree a t
  Fork : t → Tree a t → Tree a t → Tree a t
```

✓

(8) **F7.** Qual seria um tipo razoável para a *flatten* desse tipo de arvores?
Defina tal *flatten* (não se preocupe com eficiência).
RESPOSTA.

lista heterogênea

```
flatten :: Tree a t → List (Either a t)
flatten (Tip v) = [v]
flatten (Fork v l r) = flatten l ++ [v] ++ flatten r
```

não sei como transformar em um vetor heterogêneo aqui porque vai tambem tem do either, mas a ideia está no corpo!

💔

(48) **E**

Considere o tipo de expressões aritméticas envolvendo apenas números e divisões inteiras:

```
data Expr
  Val : Int → Expr
  Div : Expr → Expr → Expr
```

não redefinir

(4) **E1.** Dada a função *div* que retorna o quociente explique qual o problema da função seguinte:

```
eval : Expr → Int
eval (Val n) = n
eval (Div u v) = div (eval u) (eval v)
```

RESPOSTA.

v ?

type error assim

Eh explode caso v = 0.

..e o eval u é irrelevante

eval u

(8) **E2.** Defina uma versão segura (total) de *div* (chame de *safediv*).
DEFINIÇÃO.

(24) **E3.** Usando tua função *safediv* defina uma melhor *eval* : Expr → ?.
DEFINIÇÃO.

(12) **E4.** Um aluno—talvez não o melhor da turma—respondeu na **E3** assim:

tipo de pure

tipo de splat

```
eval : Expr → Maybe Int
eval (Val n)   = pure n
eval (Div u v) = pure safediv <*> eval u <*> eval v
```

Compila? Se não, explique o porquê. Se sim, explique porque seria melhor evitar esta definição.
RESPOSTA.

Não...

Só isso mesmo.