(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

```
Functor
fmap :: (α -> β) -> fα -> fβ

Leis:
fmap id = id
fmap (f . g) = fmap f . fmap g

Applicative:
pure :: α -> fα
(<*>) :: (fα -> fβ) -> fα -> fβ
```

F(α→β)

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b        join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.
RESPOSTA. Não podes usar notação-do.

(24) **F3.** Instancie Maybe e List como Functors, Applicatives, e Monads[3].
Sobre o List: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE Maybe.

```
Functor:
fmap _ Nothing = Nothing
fmap Just x f = Just $ f x

Applicative:
pure = Just
Just f <*> Just x = Just $ f x
_ <*> _ = Nothing

Monad
(>>=)?
```

SOBRE List. (Dadas as funções da U1.)

```
Functor:
fmap = map

Applicative:
pure x = [x]
[] <*> _ = []
_ <*> [] = []
(f : fs) <*> (x : xs) = (fmap f x ++ (fs <*> xs)

Monad
(>>=)?
```

(6) **F4.** ... e descreva **curtamente** a *interpretação computacional* de cada.[4]

(12) **F5.** Defina a *composição Kleisli* (denotada por $(>=>)$ em Haskell), com tipo:

$$(>=>) : \text{Monad } m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

```
(f >=> g) x = g >>= (f x)
```

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R                          type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) fmap : ? (conta pontinhos) fold : $(\alpha \to \alpha \to \alpha) \to \mathsf{Tree}\ \alpha \to \alpha$

(4×) forks, depth, tips : $\mathsf{Tree}\ \alpha \to \mathsf{Nat}$ (profundidade) flat : $\mathsf{Tree}\ \alpha \to \mathsf{List}\ \alpha$

(6×) search : $\alpha \to \mathsf{Tree}\ \alpha \to \mathsf{List\ Path}$ fetch : $\mathsf{Path} \to \mathsf{Tree}\ \alpha \to \mathsf{Maybe}\ \alpha$

↳ procura um valor na árvore → todos os caminhos → retornar o valor em um path

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap f (Tip x)          = Tip $ f x
fmap f (Fork (Tree x) (Tree y)) = Fork (fmap f (Tree x)) (fmap f (Tree y))

fold op (Fork (Tip x) (Tip y)) = op x y
fold op (Fork (Tree x) (Tree y)) = op (fold op Tree x) (fold op Tree y)

forks (Fork (Tip _) (Tip _)) = Succ Zero
forks (Fork (Tree x) (Tip _)) = Succ (forks (Tree x))
forks (Fork (Tip _) (Tree y)) = Succ (forks (Tree y))
forks (Fork (Tree x) (Tree y)) = Succ (forks (Tree x) + forks (Tree y))
```

Não tem data constructor Tree!

(Tip x) ??

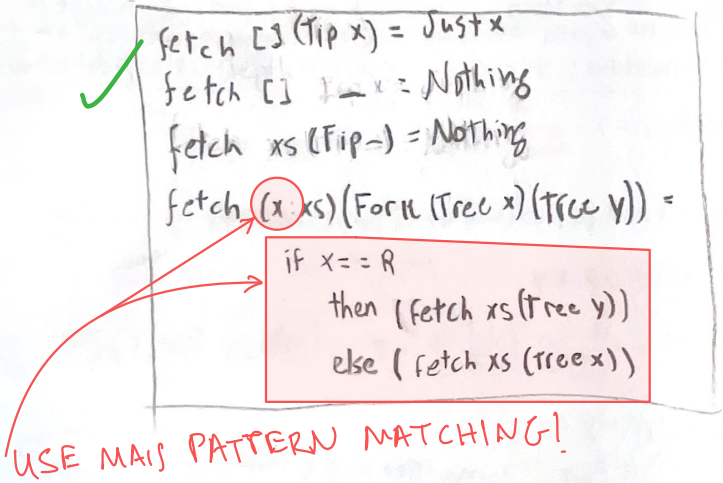
fetch no verso tips no verso

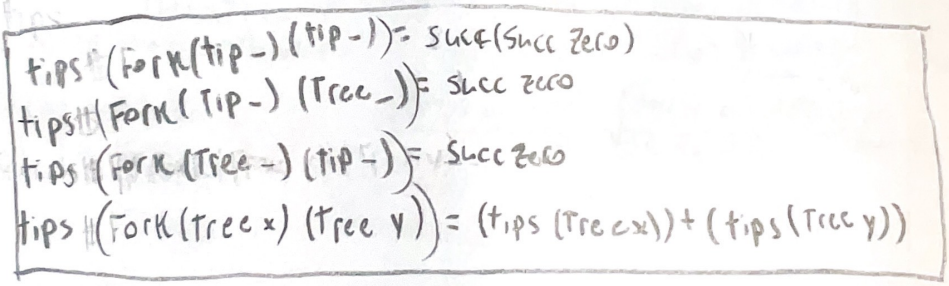
(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições devem começar com o nome da função, seguido por um '='. Podes usar `where` e lambdas se quiser — mas *sem trollagens!*[5]

```
fetch [] (Tip x) = Just x
fetch [] _       = Nothing
fetch xs (Tip _) = Nothing
fetch (x:xs) (Fork (Tree x) (Tree y)) = if x == R
                                        then
```

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.


fetch [] (Tip x) = Just x
fetch [] _ = Nothing
fetch xs (Tip _) = Nothing
fetch (x:xs) (Fork (Tree x) (Tree y)) =
if x == R
then (fetch xs (Tree y))
else (fetch xs (tree x))
USE MAIS PATTERN MATCHING!


tips (Fork (Tip _) (Tip _)) = Succ (Succ Zero)
tips (Fork (Tip _) (Tree _)) = Succ Zero
tips (Fork (Tree _) (Tip _)) = Succ Zero
tips (Fork (Tree x) (Tree y)) = (tips (Tree x)) + (tips (Tree y))

(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

```
class Functor f where
  fmap :: (a->b) -> fa -> fb

class Functor F => Applicative F
  pure :: a -> fa
  (<*>) :: f(a->b) -> fa -> Fb
```

1. `fmap id = id`.
2. `fmap (f . g) = fmap f . fmap g`.

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b        join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.

RESPOSTA. Não podes usar notação-do.

```
bind a f = join (fmap f a).
join = (>>= id).
```

(24) **F3.** Instancie `Maybe` e `List` como Functors, Applicatives, e Monads[3].
Sobre o `List`: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE `Maybe`.

```
Instance Maybe . Functor f.
  fmap f Nothing = Nothing
  fmap f (Just a) = Just (f a)
Applicative f:
  Pure = Just
  _ <*> Nothing = Nothing
  (Just f) <*> Just a = Just (f a)
  Nothing <*> Just a = ?
>>=
  return = Just
  Just x >>= m = Just (m x)
```

SOBRE `List`. (Dadas as funções da U1.)

```
Instance Functor [] where
  fmap = map
Instance Applicative [] where
  pure x = [x]
  [] <*> _ = []
  (f:fs) <*> xs = (fmap f xs) : (fs <*> xs)
Applicative ZipList [x]
  pure = ZipList ?. repeat
  (ZipList f) <*> (ZipList x) = ZipList (
    zipWith ($) f x)
-> Monads []
  return = pure
  [x:s] >>= f = f x s
```

(6) **F4.** ...e descreva **curtamente** a *interpretação computacional* de cada.[4]

`[]` -> listas como tentativas ?

`ZipList` -> listas como contexto computacional. ?

(12) **F5.** Defina a *composição Kleisli* (denotada por `(>=>)` em Haskell), com tipo:

$$(\geq\!\!>) : \mathsf{Monad}\ m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

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

return f

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

(54) **T**

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R                          type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) fmap : ? — fold : $(\alpha \to \alpha \to \alpha) \to \mathsf{Tree}\ \alpha \to \alpha$

(4×) forks, depth, tips : $\mathsf{Tree}\ \alpha \to \mathsf{Nat}$ — flat : $\mathsf{Tree}\ \alpha \to \mathsf{List}\ \alpha$

(6×) search : $\alpha \to \mathsf{Tree}\ \alpha \to \mathsf{List\ Path}$ — fetch : $\mathsf{Path} \to \mathsf{Tree}\ \alpha \to \mathsf{Maybe}\ \alpha$

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap :: (a->b) -> Tree a -> Tree b .
fmap F (Tip a) = Tip (F a)
fmap F (Fork Lt Rt) = Fork (fmap F Lt) (fmap F Rt)

flat (Tip a) = [a]
flat (Fork Lt Rt)
  = (flat Lt) ++ (flat Rt)

fold F (Tip a) = a
fold F (Fork Lt Rt) = F (fold F Lt) (fold F Rt)

tips (Tip a) = 1
tips (Fork Lt Rt) = (tips Lt) + (tips Rt)

forks (Tip a) = 0
forks (Fork Lt Rt) = 1 + (forks Lt) + (forks Rt)

fetch [] (Fork _ _) = Nothing
fetch [] (Tip a) = Just a
fetch (P s) (Tip a) = Nothing
fetch (P:Ps) (Fork Lt Rt) = case P of
                              R = fetch Ps Rt
                              L = fetch Ps Lt
```

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições devem começar com o nome da função, seguido por um '='. Podes usar `where` e λambdas se quiser — mas *sem trollagens!*[5]

```
tips = fold (+) . fmap (\x -> x / x)
```

!!

Essa é uma maneira muito complicada para descrever o número 1.
Compare com essa: 1.
Also: Funciona apenas para α numéricos
Also: Até para numéricos vai explodir no 0.

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem

(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

```
typeclass Functor where
fmap :: Functor f => (a->b)
                  -> fa -> fb
```

```
typeclass Functor => Applicative
where (<*>) :: Functor f => f(a->b)
                         -> fa -> fb
```

· Leis do Functor: fmap id = id, fmap f · fmap g = fmap (f · g)

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b
```

```
join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.

RESPOSTA. Não podes usar notação-do.

```
· join = (flip bind) id
  ou só  join mma = bind mma id
```

(24) **F3.** Instancie `Maybe` e `List` como Functors, Applicatives, e Monads[3].
Sobre o `List`: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE `Maybe`.

```
instance Functor Maybe where
  fmap :: (a->b) -> Maybe a -> Maybe b
  fmap f Nothing = Nothing
  fmap f (Just x) = Just $ f x
```

```
instance Applicative Maybe where
  Just f <*> Just x = Just $ f x
  _      <*> _      = Nothing
```

```
instance Monad Maybe where
  join Just (Just x) = Just x
  join _             = Nothing
```

SOBRE `List`. (Dadas as funções da U1.)

```
instance Functor List where
  fmap :: (a->b) -> [a] -> [b]
  fmap = map
```

```
instance Applicative List where
  ax:axs <*> xs = fmap ax xs
                  ++ axs <*> xs
  []     <*> _  = []
```

```
instance Applicative ZipList where
  axs <*> xs = zipWith ($) axs xs
```

```
instance Monad List where
  join = concat
```

(6) **F4.** ...e descreva **curtamente** a *interpretação computacional* de cada.[4]

A interpretação List é não deterministica e computa todos os resultados possíveis, um cross product. A ZipList considera que a ordem vale e computa os resultados na ordem dos pares.

(12) **F5.** Defina a *composição Kleisli* (denotada por $(\gg\!=)$ em Haskell), com tipo:

$$(\gg\!=) : \text{Monad } m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

```
(>=>) af ag x = do
  x' <- af x
  g x'
```

```
(>=>) af ag x = do
  x' <- af x
  ag x'
```

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.

[4] Deixe claro qual é qual!

(54) **T**

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R                          type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) $\mathsf{fmap} : ?$ $\qquad \mathsf{fold} : (\alpha \to \alpha \to \alpha) \to \mathsf{Tree}\ \alpha \to \alpha$

(4×) $\mathsf{forks}, \mathsf{depth}, \mathsf{tips} : \mathsf{Tree}\ \alpha \to \mathsf{Nat}$ $\qquad \mathsf{flat} : \mathsf{Tree}\ \alpha \to \mathsf{List}\ \alpha$

(6×) $\mathsf{search} : \alpha \to \mathsf{Tree}\ \alpha \to \mathsf{List\ Path}$ $\qquad \mathsf{fetch} : \mathsf{Path} \to \mathsf{Tree}\ \alpha \to \mathsf{Maybe}\ \alpha$

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap :: (a->b) -> Tree a -> Tree b
fmap f (Tip x) = Tip $ f x
fmap f (Fork tl tr) = Fork (fmap f tl) (fmap f tr)

fold f (Tip x) = x
fold f (Fork tl tr) =
  (fold f tl) `f` (fold f tr)

forks (Tip _) = 0
forks (Fork tl tr) = S (forks tl + forks tr)

tips (Tip _) = S0
tips (Fork tl tr)
  = tips tl + tips tr

depth (Tip _) = S0 0
depth (Fork tl tr) = max S(depth tl) S(depth tr

flat (Tip x) = [x]
flat (Fork tl tr) = flat tl ++ flat tr

fetch [] (Tip x) = Just x
fetch [] (Fork _ _) = Nothing
fetch (x:xs) (Tip _) = Nothing

fetch (x:xs) (Fork tl tr)
  = case x of
      L -> fetch tl xs
      R -> fetch tr xs
```

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições devem começar com o nome da função, seguido por um '='. Podes usar `where` e λambdas se quiser — mas *sem trollagens!*[5]

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.

(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

```
Functor
fmap :: (a -> b) -> f a -> f b

Applicative
pure :: a -> f a
<*> :: f (a -> b) -> f a -> f b

Leis de Functor
fmap id = id
fmap (f . g) = fmap f . fmap g
```

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b        join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.
RESPOSTA. Não podes usar notação-do.

```
bind mx f = join (fmap f mx)

join mmx = bind mmx (\x -> x)
```

id

(24) **F3.** Instancie Maybe e List como Functors, Applicatives, e Monads[3].
Sobre o List: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE Maybe.

```
Functor
fmap _ Nothing = Nothing
fmap f (Just x) = Just (f x)

Applicative
pure x = Just x
Nothing <*> _ = Nothing
_ <*> Nothing = Nothing
(Just f) <*> (Just x) = Just (f x)

Monad
bind Nothing _ = Nothing
bind (Just x) f = Just (f x)
```

SOBRE List. (Dadas as funções da U1.)

```
fmap = map

pure x = repeat x
pure x = ZipList (repeat x)

[] <*> _ = []
_ <*> [] = []
(f:fs) <*> xs = fmap f xs ++ (fs <*> xs)
ZipList (f:fs) <*> ZipList (x:xs) = f x : (ZipList fs <*> ZipList xs)

bind xs f = concat (fmap f xs)
```

join = concat

(6) **F4.** ...e descreva **curtamente** a *interpretação computacional* de cada.[4]

A interpretação dos Listas no operador <*> é de produto cartesiano
A interpretação dos ZipList no operador <*> é de aplicar a função no operador de mesmo index

(12) **F5.** Defina a *composição Kleisli* (denotada por (>=>) em Haskell), com tipo:

$$(>=>) : \text{Monad } m \Rightarrow (a \to m\ b) \to (b \to m\ c) \to (a \to m\ c)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

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

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

T

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R                          type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) fmap : ? — fold : $(\alpha \to \alpha \to \alpha) \to \text{Tree } \alpha \to \alpha$

(4×) forks, depth, tips : $\text{Tree } \alpha \to \text{Nat}$ — flat : $\text{Tree } \alpha \to \text{List } \alpha$

(6×) search : $\alpha \to \text{Tree } \alpha \to \text{List Path}$ — fetch : $\text{Path} \to \text{Tree } \alpha \to \text{Maybe } \alpha$

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap :: (a->b) -> Tree a -> Tree b
fmap f (Tip x) = Tip (f x)
fmap f (Fork lt rt) = Fork (fmap f lt) (fmap f rt)

tips Tip _ = 1
tips (Fork lt rt) = (tips lt)+(tips rt)

forks Tip _ = 0
forks (Fork lt rt) = 1 + (forks lt) + (forks rt)

depth Tip _ = 0
depth (Fork lt rt) = 1 + (if dlt > drt then dlt else drt)
  where dlt = depth lt
        drt = depth rt

flat Tip x = [x]
flat (Fork lt rt) = (flat lt) ++ (flat rt)

fold f (Tip x) = x
fold f (Fork lt rt) = f (fold f lt) (fold f rt)
```

max dlt drt

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições **devem** começar com o nome da função, seguido por um '='. Podes usar `where` e λambdas se quiser — mas *sem trollagens!*[5]

```
tips = (fold (+)) . (fmap (\_->1))

flat = (fold (++)) . (fmap (\x -> [x]))
```

(++)

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.

(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

```
Functor:
  fmap :: (a -> b) -> f a -> f b

Applicative
  pure :: a -> f a
  <*> :: f(a -> b) -> f a -> f b
```

fmap id = id | fmap (f . g) = fmap f . fmap g

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b        join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.
RESPOSTA. Não podes usar notação-do.

```
bind
```

(24) **F3.** Instancie `Maybe` e `List` como Functors, Applicatives, e Monads[3].
Sobre o `List`: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE `Maybe`.

```
Functor:
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just $ f x
Applicative:
  pure = Just
  (Just f) <*> (Just x) = Just $ f x
  _ <*> _ = Nothing
Monad:
  Nothing >>= _ = Nothing
  (Just x) >>= f = f x
```

SOBRE `List`. (Dadas as funções da U1.)

```
Functor #:
  fmap = map
Applicative 1:
  pure x = [x]          repeat x
  (f:fs) <*> (x:xs) = f x : (fs <*> xs)
  _ <*> _ = []          zipWith ($)
Applicative 2:
  pure x = [x]
  (f:fs) <*> xs = (fmap f xs) ++ (fs <*> xs)
  _ <*> _ = []          cp
Monad:
  join = concat
```

(6) **F4.** ... e descreva **curtamente** a *interpretação computacional* de cada.[4]

O modo 1: listas vistas como sequências de valores no "tempo".
O modo 2: listas vistas como valores não-deterministas.

(12) **F5.** Defina a *composição Kleisli* (denotada por $(\gg\!=)$ em Haskell), com tipo:

$$(\gg\!=) : \text{Monad } m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

```
(>=>) f g x = bind (f x) g
```

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

(f >=> g) x =

(51) **T**

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R                          type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) fmap : ? fold : (α → α → α) → Tree α → α

(4×) forks, depth, tips : Tree α → Nat flat : Tree α → List α

(6×) search : α → Tree α → List Path fetch : Path → Tree α → Maybe α

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap :: (a → b) → Tree a → Tree b
fmap f (Fork x y) = Fork (fmap f x) (fmap f y)
fmap f (Tip x) = Tip $ f x

flat (Tip x) = [x]
flat (Fork x y) = (flat x) ++ (flat y)

tips (Tip x) = 1
tips (Fork x y) = (tips x) + (tips y)

forks (Fork x y) = 1 + (forks x) + (forks y)
forks (Tip x) = 0

depth (Fork x y) = 1 + max (depth x) (depth y)
depth (Tip x) = 0
```

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições **devem** começar com o nome da função, seguido por um '='. Podes usar where e λambdas se quiser — mas *sem trollagens!*[5]

```
tips = fold (+) 0 (fmap (const 1))
```

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.

(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

```
Functor :: * -> *
Applicative :: * -> *
```

Leis de functor:
fmap id = id
fmap (f . g) = fmap f . fmap g

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b
join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.
RESPOSTA. Não podes usar notação-do.

(24) **F3.** Instancie Maybe e List como Functors, Applicatives, e Monads[3].
Sobre o List: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE Maybe.

```
Instance Functor Where
  fmap :: (α → β) → Maybe α → Maybe β
  fmap _ Nothing = Nothing
  fmap f (Maybe x) = Maybe (f x)
```

Just Just

SOBRE List. (Dadas as funções da U1.)

```
Instance Functor where
  fmap :: (α → β) → List α → List β
  fmap = map
```

(6) **F4.** ...e descreva **curtamente** a *interpretação computacional* de cada.[4]

(12) **F5.** Defina a *composição Kleisli* (denotada por (>=>) em Haskell), com tipo:

$$(\gg\!=\!\gg) : \text{Monad } m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

1) **T**

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R
type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) fmap : ? ✓ fold : $(\alpha \to \alpha \to \alpha) \to \text{Tree } \alpha \to \alpha$ ✓

(4×) forks, depth, tips : $\text{Tree } \alpha \to \text{Nat}$ ✓ flat : $\text{Tree } \alpha \to \text{List } \alpha$ ✓

(6×) search : $\alpha \to \text{Tree } \alpha \to \text{List Path}$ fetch : $\text{Path} \to \text{Tree } \alpha \to \text{Maybe } \alpha$ ✓

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap :: (α → β) → Tree α → Tree β
fmap Tip x = Tip (f x)
fmap (Fork x y) = Fork (fmap x) (fmap y)

depth (Tip x) = 1 0
depth (Fork x y) = if (S(depth x) > S(depth y))
                   then S(depth x) else S(depth y)
```

use max!

```
tips (Tip x) = 1
tips (Fork x y) = (tips x) + (tips y)

forks (Tip x) = 0
forks (Fork x y) = S((forks x) + (forks y))

flat (Tip x) = [x]
flat (Fork x y) = (flat x) ++ (flat y)

fold _ (Tip x) = x
fold op (Fork x y) = op (fold op x) (fold op y)
```

Mais definições no verso...

(18) **T2.** Mostre como definir as tips, depth, flat como composições em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições **devem** começar com o nome da função, seguido por um '='. Podes usar `where` e λambdas se quiser — mas *sem trollagens!*[5]

```
tips = fold (+) . fmap (\x → 1)
depth ≈ max . ++ . fmap (fold (+))   type error
flat = fold (++) . fmap (\x → [x])
```

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.

(66) **F**

(12) **F1.** Defina as typeclasses Functor e Applicative, e enuncie as leis de Functor.

```
class Functor F where
  fmap :: (a -> b) -> Fa -> Fb

Leis de Functor
  fmap id = id
  fmap (f.g) = fmap f . fmap g

class Applicative F where
  pure :: a -> Fa
  (<*>) :: F(a->b) -> Fa -> Fb
```

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b        join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.
RESPOSTA. Não podes usar notação-do.

```
bind mx f = join $ fmap f mx

join mmx = bind mmx id
```

(24) **F3.** Instancie Maybe e List como Functors, Applicatives, e Monads[3].
Sobre o List: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE Maybe.

```
Functor
fmap f (Just x) = Just $ f x
fmap _ _ = Nothing

Applicative
pure = Just
(<*>) (Just f) (Just x) = Just $ f x
(<*>) _ _ = Nothing

Monad
join (Just x) = x
join _ = Nothing
```

SOBRE List. (Dadas as funções da U1.)

```
Functor                          Monad
fmap = map                       join [] = []; join (x:xs) = x ++ join xs

App (1)
pure x = [x]
(<*>) [] _ = []
(<*>) _ [] = []
(<*>) (F:Fs) Xs =
  fmap F Xs ++ (Fs <*> Xs)

App (2)
pure x = [x]   repeat x
(<*>) [] _ = []
(<*>) _ [] = []
(<*>) (F:Fs) (X:Xs) =
  F X : (Fs <*> Xs)
```

concat

(6) **F4.** ...e descreva **curtamente** a *interpretação computacional* de cada.[4]

(1) INDETERMINÍSTICA, IRÁ REALIZAR TODAS AS COMBINAÇÕES POSSÍVEIS
(2) DETERMINÍSTICA, APLICARÁ A F_1 NO X_1 E SERÁ LIMITADO PELO NÚMERO

sequencial/temporal/paralela

(12) **F5.** Defina a *composição Kleisli* (denotada por (>=>) em Haskell), com tipo:

$$(>=>) : \text{Monad } m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

```
(>=>) Fmb Fmy a =  bind (bind (pure a) Fmb) Fmy
```

!!

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

fold

T

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R                          type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) fmap : ? — fold : $(\alpha \to \alpha \to \alpha) \to \mathsf{Tree}\ \alpha \to \alpha$

(4×) forks, depth, tips : $\mathsf{Tree}\ \alpha \to \mathsf{Nat}$ — flat : $\mathsf{Tree}\ \alpha \to \mathsf{List}\ \alpha$

(6×) search : $\alpha \to \mathsf{Tree}\ \alpha \to \mathsf{List\ Path}$ — fetch : $\mathsf{Path} \to \mathsf{Tree}\ \alpha \to \mathsf{Maybe}\ \alpha$

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap :: (a -> b) -> Tree a -> Tree b
fmap f (Tip x)  = Tip $ f x
fmap f (Fork l r) = Fork (fmap f l) (fmap f r)

fold f (Tip x)  = f x x
fold f (Fork l r) = f (fold l) (fold r)
?

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

depth (Tip _) = 0
depth (Fork l r) | depth l > depth r = S (depth l)
                 | otherwise = S (depth r)

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

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

fetch [] (Tip x) = Just x
fetch [] _ = Nothing
fetch (p:ps) (Tip _) = Nothing
fetch (p:ps) (Fork l r) | p == L = fetch ps l
                        | otherwise = fetch ps r
```

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições **devem** começar com o nome da função, seguido por um '='. Podes usar `where` e λambdas se quiser — mas *sem trollagens!*[5]

```
tips = (fold (+) 0 . fmap (\x -> if isTip l then tips (g l x) + tips (g r x) |
  where
    g l (Fork l _) = l
    g r (Fork _ r) = r
```

?!

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.

(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

```
Functor:
fmap:: Functor f => (a->b) -> f a -> f b
Leis: fmap id = id
      fmap (g . f) = fmap g . fmap f
```

```
Applicative
pure:: Applicative f => a -> f a
(<*>):: Applicative f => f(a->b) -> fa -> fb
```

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b        join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.
RESPOSTA. Não podes usar notação-do.

```
bind mx f_my = join (fmap f_my mx)
join mmx = bind mmx id
```

(24) **F3.** Instancie Maybe e List como Functors, Applicatives, e Monads[3].
Sobre o List: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE Maybe.

```
Functor
fmap f (Just x) = Just (f x)
fmap _ Nothing = Nothing

Applicative
pure = Just
(Just f) <*> (Just x) = Just (f x)
_ <*> _ = Nothing

Monad
bind (Just x) f = f x
bind Nothing _ = Nothing
```

SOBRE List. (Dadas as funções da U1.)

```
Functor
fmap = map

Applicative 1:
pure x = [x]
[] <*> xs = []
(f:fs) <*> xs = fmap f xs ++ fs <*> xs

Applicative 2
pure = repeat
(<*>) = zipWith ($)

Monad:
join = concat
```

(6) **F4.** ...e descreva **curtamente** a *interpretação computacional* de cada.[4]

A lista com applicative 1 representa um conjunto não determinístico de coisas e para combinarmos duas destas listas é necessário combinar todos os elementos de todas as formas possíveis. A lista com applicative 2 os seus elementos combinados de acordo com as posições: 1º com 1º, 2º com 2º, ...

(12) **F5.** Defina a *composição Kleisli* (denotada por `(>=>)` em Haskell), com tipo:

$$(>=>) : \text{Monad } m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

```
f_my >=> f_mz = \x -> f_my x >>= f_mz
```

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R
                                        type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(1×) fmap : ? — fold : (α → α → α) → Tree α → α

(4×) forks, depth, tips : Tree α → Nat — flat : Tree α → List α

(6×) search : α → Tree α → List Path — fetch : Path → Tree α → Maybe α

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap : (a→b) → Tree a → Tree b
fmap f (Tip x) = Tip (f x)
fmap f (Fork ll lr) =
    Fork (fmap f ll) (fmap f lr)

forks (Fork ll lr) = 5 (forks ll + forks lr)
forks _            = 0

tips (Tip _) = 50
tips (Fork ll lr) = tips ll + tips lr

depth (Tip _ x) = 0
depth (Fork ll lr) =
    1 + (max (depth ll) (depth lr))

flat (Tip _ x) = [x]
flat (Fork ll lr) = flat ll ++ flat lr

fold f (Tip x) = x
fold f (Fork ll lr) = f (fold f ll) (fold f lr)
```

```
fetch [] (Tip x) = Just x
fetch (x:xs) (Fork ll lr) =
  case x of
    L → fetch xs ll
    R → fetch xs lr
fetch _ _ = Nothing

search x (Fork ll lr) =
```

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições devem começar com o nome da função, seguido por um '='. Podes usar where e λambdas se quiser — mas *sem trollagens*![5]

```
forks = fold (+) . fmap g
  where g (Tip x) = Tip 0
        g (Fork ll lr) =
            5 (Fork ll lr)
```

```
tips = fold (+) . fmap f
  where f (Tip _) = Tip (50)
        f x       = x
```

Type error

```
depth = fold max . fmap g
  where g (Tip x) =
```

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.

(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

class Functor f where
  fmap :: (α → β) → f α → f β
class Applicative f where
  pure :: α → f α
  <*> :: f (α → β) → f α → f β

leis do functor:
1: (fmap . Id) x = x
2º fmap (f.g) x = fmap f (fmap g x)

point-free!

(12) **F2.** Um applicative `m :: * -> * ` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b
join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.

RESPOSTA. Não podes usar notação-do.

(24) **F3.** Instancie Maybe e List como Functors, Applicatives, e Monads[3].
Sobre o List: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE Maybe.

instance Functor Maybe where
  fmap _ Nothing = nothing
  fmap f (Just x) = Just (f x)
instance Applicative Maybe where
  pure x = Just x
  = Just (f x)
Just f <*> Just x =

SOBRE List. (Dadas as funções da U1.)

instance Functor [] where
  fmap = map
instance Applicative [] where
  pure x = [x]
(I) (f:fs) <*> (x:xs) =
  f x : (fs <*> xs)
ou
(II) (f:fs) <*> xs = fmap f xs : (fs <*> xs)

(6) **F4.** ...e descreva **curtamente** a *interpretação computacional* de cada.[4]

I – equivale a aplicar f elemento a elemento
II – faz a fazer um distributividade em todos

(12) **F5.** Defina a *composição Kleisli* (denotada por (>=>) em Haskell), com tipo:

$$(>=>) : \text{Monad } m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

[3] Podes usar tanto a definição baseada no bind quanto no join.
[4] Deixe claro qual é qual!

T

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R                                   type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) fmap : ? fold : $(\alpha \to \alpha \to \alpha) \to \mathsf{Tree}\ \alpha \to \alpha$.

(4×) forks, depth, tips : $\mathsf{Tree}\ \alpha \to \mathsf{Nat}$ flat : $\mathsf{Tree}\ \alpha \to \mathsf{List}\ \alpha$

(6×) search : $\alpha \to \mathsf{Tree}\ \alpha \to \mathsf{List\ Path}$ fetch : $\mathsf{Path} \to \mathsf{Tree}\ \alpha \to \mathsf{Maybe}\ \alpha$

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap f (Tip x) = Tip (f x)
fmap f (Fork z w) = Fork (fmap f z) (fmap f w)

forks Tip _ = 0
forks (Fork x y) = 1 + forks x + forks y

tips Tip _ = 1
tips (Fork x y) = tips x + tips y

depth Fork (Tip _) (Tip _) = 0 max
depth (Fork x y) = 1 + depth x + depth y

fold f (Tip x) = x
fold f (Fork x y) = (fold f x) `f` (fold f y)

flat (Tip x) = [x]
flat (Fork z w) = flat z [++] flat w
```

```
fetch (Tip x) [] = Nothing = Nothing
fetch _ (_:_) = Nothing = sujeira
fetch (x:xs) (Fork x y) =
  if x == L then fetch xs x
  else fetch xs y
```

search

PATTERN MATCHING UÉ!

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições **devem** começar com o nome da função, seguido por um '='. Podes usar `where` e λambdas se quiser — mas *sem trollagens!*[5]

```
depth = fold . (fmap g) -1
  where g = λx → (+) 1

tips = fold . (fmap g)
  where g = λx → (+) 1

flat = fold (:)
```

esqueceu o argumento da fold?

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.

(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

```
f Functor :: * -> *
fmap :: (α -> β) -> f α -> f β
```

```
f Applicative : f Functor
<*> :: f(α->β) -> fα -> fβ
pure :: α -> f α
```

Leis de Functor:
fmap id = id
fmap (f . g) = fmap f . fmap g

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b        join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.
RESPOSTA. Não podes usar notação-do.

```
bind ax f = join (fmap f ax)        join ax = bind ax id
```

(24) **F3.** Instancie Maybe e List como Functors, Applicatives, e Monads[3].
Sobre o List: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE Maybe.

```
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)

pure = Just
(Just f) <*> (Just x) = Just (f x)
_ <*> _ = Nothing

Nothing >>= f = Nothing
(Just x) >>= f = f x
```

SOBRE List. (Dadas as funções da U1.)

```
fmap = map
pure x = [x]                    (1)
pure x = x : repeat x           (2)

<*> [] _ = []
<*> (f:fs) xs = (map f xs) ++ (<*> fs xs)   } (1)
<*> = zipWith ($)                          (2)

(>>=) xs f = concat (map f xs)
```

(6) **F4.** ...e descreva **curtamente** a *interpretação computacional* de cada.[4]

```
A (1) é não determinística, faz todas as combinações possíveis entre os elem
A (2) aplica as funções da lista apenas nos termos de posição correspond
```

(12) **F5.** Defina a *composição Kleisli* (denotada por $(\gg\!=)$ em Haskell), com tipo:

$$(\gg\!=) : \text{Monad } m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

```
(f >=> g) x = bind (f x) g
```

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

T

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R                                   type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) $\mathsf{fmap} : ?\ (\alpha \to \beta) \to \mathsf{Tree}\ \alpha \to \mathsf{Tree}\ \beta$ $\quad \mathsf{fold} : (\alpha \to \alpha \to \alpha) \to \mathsf{Tree}\ \alpha \to \alpha$

(4×) $\mathsf{forks}, \mathsf{depth}, \mathsf{tips} : \mathsf{Tree}\ \alpha \to \mathsf{Nat}$ $\quad \mathsf{flat} : \mathsf{Tree}\ \alpha \to \mathsf{List}\ \alpha$

(6×) $\mathsf{search} : \alpha \to \mathsf{Tree}\ \alpha \to \mathsf{List\ Path}$ $\quad \mathsf{fetch} : \mathsf{Path} \to \mathsf{Tree}\ \alpha \to \mathsf{Maybe}\ \alpha$

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap f (Tip x) = Tip (f x)
fmap f (Fork l r) = Fork (fmap f l) (fmap f r)

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

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

fold f (Tip x) = x
fold f (Fork l r) = f (fold f l) (fold f r)

depth (Tip _) = S 0
depth (Fork l r) = S (max (depth l) (depth r))
    where max x y = if x > y then x else y

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

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições **devem** começar com o nome da função, seguido por um '='. Podes usar `where` e λambdas se quiser — mas *sem trollagens!*[5]

```
tips = fold (+) . fmap (const 1)

depth =

flat = fold (++) . fmap (λx. [x])
```

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.

(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

Functor: fmap :: $(\alpha \to \beta) \to f\,\alpha \to f\,\beta$
fmap · id = id
fmap (f · g) = (fmap f) · (fmap g)

Applicative:
pure: $\alpha \to f\,\alpha$
(<*>): $f(\alpha \to \beta) \to f\,\alpha \to f\,\beta$

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b        join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.
RESPOSTA. Não podes usar notação-do.

(24) **F3.** Instancie `Maybe` e `List` como Functors, Applicatives, e Monads[3].
Sobre o `List`: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE `Maybe`.

```
fmap _ Nothing = Nothing
fmap f (Just x) = Just (f x)

pure = Just

(<*>) f (Just x) =
```

SOBRE `List`. (Dadas as funções da U1.)

```
fmap = map

pure = repeat
```

(6) **F4.** "...e descreva **curtamente** a *interpretação computacional* de cada.[4]

(12) **F5.** Defina a *composição Kleisli* (denotada por $(>\!=\!>)$ em Haskell), com tipo:

$$(>\!=\!>) : \text{Monad } m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

```
(f >=> g) x = bind (f x) g
```

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

(54) # T

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R                                   type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) $\mathsf{fmap} : ?$ $\qquad$ $\mathsf{fold} : (\alpha \to \alpha \to \alpha) \to \mathsf{Tree}\ \alpha \to \alpha$

(4×) $\mathsf{forks}, \mathsf{depth}, \mathsf{tips} : \mathsf{Tree}\ \alpha \to \mathsf{Nat}$ $\qquad$ $\mathsf{flat} : \mathsf{Tree}\ \alpha \to \mathsf{List}\ \alpha$

(6×) $\mathsf{search} : \alpha \to \mathsf{Tree}\ \alpha \to \mathsf{List\ Path}$ $\qquad$ $\mathsf{fetch} : \mathsf{Path} \to \mathsf{Tree}\ \alpha \to \mathsf{Maybe}\ \alpha$

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap :: (α → β) → Tree α → Tree β
fmap f Tip x = Tip fx
fmap f (Fork l r) = Fork (fmap l) (fmap r)

fold op Tip x = x
fold op (Fork l r) = (fold op l) `op` (fold op r)

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

forks Tip x = 0
forks (Fork l r) = S((forks l) + (forks r))

tips Tip x = S0
tips (Fork l r) = (tips l) + (tips r)

depth Tip x = 0
depth (Fork l r) = S(max (depth l, depth r))

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

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições **devem** começar com o nome da função, seguido por um '='. Podes usar `where` e λambdas se quiser — mas *sem trollagens!*[5]

```
flat = fold (++) fmap (_:[])
```

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.

(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

```
class Functor F where
  fmap :: (a -> b) -> f a -> f b
  -- fmap ida = ida
  -- fmap (f.g) = fmap f . (fmap g)

class Functor f => Applicative where
  (<$>) :: f (a -> b) -> fa -> fb
  pure :: a -> fa
```

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b        join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.
RESPOSTA. Não podes usar notação-do.

```
bind ax fa = join (fmap fa ax)
```

(24) **F3.** Instancie `Maybe` e `List` como Functors, Applicatives, e Monads[3].
Sobre o `List`: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE `Maybe`.

```
instance Functor Maybe where
  fmap f (Just x) = Just (fx)
  fmap _ _        = Nothing

instance Applicative Maybe where
  pure x = Just x
  (<$>) (Just f) (Just x) = Just (fx)
  (<$>) _ _ = Nothing
```

SOBRE `List`. (Dadas as funções da U1.)

```
instance Functor List where
  fmap = map

instance Applicative List where
  pure x = [x]
  (<$>) [] _ = []
  (<$>) (f:fs) xs = map f xs ++ (fs <$> xs)

instance Applicative List where
  pure x = [x]
  (<$>) [] _ = []
  (<$>) (f:fs) (x:xs) = f x : (fs <$> xs)
```

repeat x

(6) **F4.** ...e descreva **curtamente** a *interpretação computacional* de cada.[4]

Na primeira versão do Applicative, é válido interpretar a lista como algo não ordenado, logo queremos aplicar todas as funções da primeira lista em todos os elementos da segunda. Na outra interpretação, podemos ver listas como algo indexado, logo aplicamos a enésima função no enésimo termo.

(12) **F5.** Defina a *composição Kleisli* (denotada por $(>=>)$ em Haskell), com tipo:

$$(>=>) : \text{Monad } m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do. mas nesse caso a questão vale metade dos pontos.)

```
f >=> g = do
  x <- f
```

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

F3.

```
instance Monad Maybe where
  join (Just x) = x
  join _ Nothing = Nothing

instance Monad List where
  join [] = []
  join (x:xs) = x ++ (join xs)
```

T

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R                          type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) fmap : ? — fold : $(\alpha \to \alpha \to \alpha) \to \mathsf{Tree}\ \alpha \to \alpha$

(4×) forks, depth, tips : $\mathsf{Tree}\ \alpha \to \mathsf{Nat}$ — flat : $\mathsf{Tree}\ \alpha \to \mathsf{List}\ \alpha$

(6×) search : $\alpha \to \mathsf{Tree}\ \alpha \to \mathsf{List\ Path}$ — fetch : $\mathsf{Path} \to \mathsf{Tree}\ \alpha \to \mathsf{Maybe}\ \alpha$

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fold op (Tip x) = x
fold op (Fork l r) = (fold op l) `op` (fold op r)

fmap f (Tip x) = Tip (f x)
fmap f (Fork l r) = Fork (fmap f l) (fmap f r)

forks (Tip x) = 0
forks (Fork l r) = 1 + (forks l) + (forks r)

depth (Fork l r) = 1 + (if (depth l) > (depth r) then depth l else depth r)
depth _ = 0

tips (Fork l r) = (tips l) + (tips r)
tips _ = 1

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

search x (Tip y) = if x == y then [[]] else []
search x (Fork l r) = map (L:) (search x l) ++ map (R:) (search x r)

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

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições **devem** começar com o nome da função, seguido por um '='. Podes usar `where` e λambdas se quiser — mas *sem trollagens!*[5]

```
flat = fold (++). fmap (\x -> [x])

tips = fold (+). fmap (\x -> 1)

depth = fold (1+ \x y -> if x > y then x else y). fmap (\x -> 0)
```

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.

```
search x (Tip y) = if x == y then [[]] else []
search x (Fork l r) = if fetch (Fork l r) = Nothing then [] else
                      map (L:)
```

(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

```
fmap :: (a→b) → f a → f b
fmap id = id        fmap (f.g) = (fmap f).(fmap g)
```

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b        join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.
RESPOSTA. Não podes usar notação-do.

```
bind ax f = f (join ax)
```

(24) **F3.** Instancie `Maybe` e `List` como Functors, Applicatives, e Monads[3].
Sobre o `List`: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE `Maybe`.

```
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing
```

SOBRE `List`. (Dadas as funções da U1.)

```
instance Functor [] where
  fmap = map
```

(6) **F4.** ...e descreva **curtamente** a *interpretação computacional* de cada.[4]

(12) **F5.** Defina a *composição Kleisli* (denotada por `(>=>)` em Haskell), com tipo:

$$(\rightarrow\!\!=\!\!\Rightarrow) : \text{Monad } m \Rightarrow (\alpha \rightarrow m\ \beta) \rightarrow (\beta \rightarrow m\ \gamma) \rightarrow (\alpha \rightarrow m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

```
f >=> g = g . join . f
```

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

T

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R                          type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) $\quad$ fmap : ? $\qquad$ fold : $(\alpha \to \alpha \to \alpha) \to \text{Tree } \alpha \to \alpha$

(4×) $\quad$ forks, depth, tips : $\text{Tree } \alpha \to \text{Nat}$ $\qquad$ flat : $\text{Tree } \alpha \to \text{List } \alpha$

(6×) $\quad$ search : $\alpha \to \text{Tree } \alpha \to \text{List Path}$ $\qquad$ fetch : $\text{Path} \to \text{Tree } \alpha \to \text{Maybe } \alpha$

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap :: (a->b) -> Tree a -> Tree b
fmap f (Tip x) = Tip (f x)
fmap f (Fork tl tr) = Fork (fmap f tl) (fmap f tr)

fold op (Tip x) = x
fold op (Fork tl tr) = op (fold op tl) (fold op tr)

forks (Tip _) = 0
forks (Fork tl tr) = 1 + forks tl + forks tr

depth (Tip _) = 1 0
depth (Fork tl tr) = max (1+depth tl) (1+depth tr)

tips (Tip _) = 1
tips (Fork tl tr) = tips tl + tips tr

flat (Tip x) = [x]
flat (Fork tl tr) = flat tl ++ flat tr

fetch [] (Tip x) = Just x
fetch [] _       = Nothing
fetch (s:_) (Tip _) = Nothing
fetch (s:ss) (Fork tl tr) =
  case s of
    -> L = fetch ss tl
    -> R = fetch ss tr
```

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições **devem** começar com o nome da função, seguido por um '='. Podes usar where e λambdas se quiser — mas *sem trollagens!*[5]

```
tips = fold (+) . fmap (const 1)
flat = fold (++) . fmap (:[])

depth = fold (max) . fmap (const h)
      where h (Tip _) = 1
            h (Fork tl tr) = (h tl) + 1
```

~ type error

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.

(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

Functor
Necessário
fmap :: Functor f => (a->b) -> f a -> f b
fmap id = id
fmap (f.g) = (fmap f).(fmap g)

Applicative
Necessário:
pure :: ? <*>

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b        join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.

RESPOSTA. Não podes usar notação-do.

```
bind x y = (\x y -
```

```
join x y =
```

(24) **F3.** Instancie `Maybe` e `List` como Functors. Applicatives. e Monads[3].
Sobre o `List`: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE `Maybe`.

```
Instance Monad Maybe where
-- join :: Maybe (Maybe a) -> Maybe a
join x = fmap (fmap id)

Instance Functor Maybe where
-- fmap :: (a->b) -> Maybe a -> Maybe b
fmap f Nothing = Nothing
fmap f x = Just (f x)

Instance Applicative Maybe where
-- pure :: Maybe a -> (a->b) -> b
pure f (Maybe x) = (\(Maybe x) -> x) . fmap f (Maybe x)
-- (<*>) :: (a->b) -> a -> b -> b
(<*>) x y = fmap (fmap x) y
```

SOBRE List. (Dadas as funções da U1.)

```
Instance Monad [] where
-- join :: [[a]] -> [a]
join [x] = foldr (++) [] x

Instance Functor [] where
-- fmap :: (a->b) -> [a] -> [b]
fmap f [] = []
fmap f (x:xs) = f x : (fmap f xs)

Instance Applicative [] where:
-- pure :: a -> [a]
pure f [x] = (\[x] -> x) . fmap f [x]
-- (<*>) :: [a->b] -> [a] -> [b]
(<*>) x y = fmap (fmap x) y
```

(6) **F4.** ...e descreva **curtamente** a *interpretação computacional* de cada.[4]

Functor é uma caixa que guarda valores (ou valor)
Applicative é um Functor sujo :(não tem nada sujo aqui!

(12) **F5.** Defina a *composição Kleisli* (denotada por $(\rightarrow)$ em Haskell), com tipo:

$$(\rightarrow) : \text{Monad } m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

T

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R                          type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) fmap : ? (conta os forks) (length α) — fold : $(\alpha \to \alpha \to \alpha) \to \mathsf{Tree}\ \alpha \to \alpha$

(4×) forks, depth, tips : $\mathsf{Tree}\ \alpha \to \mathsf{Nat}$ — flat : $\mathsf{Tree}\ \alpha \to \mathsf{List}\ \alpha$

(6×) search : $\alpha \to \mathsf{Tree}\ \alpha \to \mathsf{List\ Path}$ (profundidade; caminho [L,R]) — fetch : $\mathsf{Path} \to \mathsf{Tree}\ \alpha \to \mathsf{Maybe}\ \alpha$

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap :: (a->b) -> Tree a -> Tree b
fmap f (tip α) = tip (f α)
fmap f (fork L R) = fork (fmap f L) (fmap f R)

flat (tip x) = [x]
flat (fork x y) = (flat x) ++ (flat y)

tips (tip _) = 1
tips (fork x y) = (tips x) + (tips y)

forks (tip _) = 0
forks (fork x y) = (1 + (forks x)) + (0 + (forks y))
```

```
fold f b (tip x) = b `f` x
fold f b (fork L R) = x `f` y
  where
  x = fold f b L
  y = fod f b R
```

o que serve somar 0?

?

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições **devem** começar com o nome da função, seguido por um '='. Podes usar where e λambdas se quiser — mas *sem trollagens!*[5]

```
tips x = (fold (+) 0) . (fmap (\x -> 1)) $ x
```

o "ponto" era sem ponto!

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.

(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
LAWS:
  fmap id = id    fmap (g . f) = fmap g . fmap f
```

```
class (*) Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
(*) Functor f =>
```

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b        join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.
RESPOSTA. Não podes usar notação-do.

```
bind mx f = join $ fmap f mx

join mmx = bind mmx id
```

(24) **F3.** Instancie `Maybe` e `List` como Functors, Applicatives, e Monads[3].
Sobre o `List`: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE `Maybe`.

```
fmap _ Nothing = Nothing
fmap f (Just x) = Just $ f x

pure = Just

Nothing <*> _ = Nothing
_ <*> Nothing = Nothing
Just f <*> Just x = Just $ f x

join Nothing = Nothing
join (Just mx) = mx
```

SOBRE `List`. (Dadas as funções da U1.)

```
fmap = map

(1) pure x = repeat x

[] <*> _ = []
_ <*> [] = []
(f:fs) <*> (x:xs) = f x : (fs <*> xs)

(2) pure x = [x]

[] <*> _ = []
(f:fs) <*> xs = fmap f xs ++ (fs <*> xs)

join = concat
```

(6) **F4.** ...e descreva **curtamente** a *interpretação computacional* de cada.[4]

```
(1) Uma sequência
(2) Um conjunto
```

(12) **F5.** Defina a *composição Kleisli* (denotada por $(\gg\!=)$ em Haskell), com tipo:

$$(\gg\!=) : \text{Monad } m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

```
(f >=> g) x = f x >>= g
```

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

Γ

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R                                   type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(1×) fmap : ? fold : (α → α → α) → Tree α → α

(4×) forks, depth, tips : Tree α → Nat flat : Tree α → List α

(6×) search : α → Tree α → List Path fetch : Path → Tree α → Maybe α

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap :: (α → β) → Tree α → Tree β
fmap f (Tip x)    = Tip (f x)
fmap f (Fork l r) = Fork (fmap f l) (fmap f r)

fold _ (Tip x)    = x
fold op (Fork l r) = fold op l `op` fold op r

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

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

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

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

search a (Tip x)
  | a == x = [[]]
  | otherwise = []
search a (Fork l r) = fmap (L:) (search a l)
                    ++ fmap (R:) (search a r)

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

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições **devem** começar com o nome da função, seguido por um '='. Podes usar `where` e λambdas se quiser — mas *sem trollagens!*[5]

```
tips  = fold (+) . fmap (\_ -> 1)
depth = fold (S . max) . fmap (\_ -> 0)
flat  = fold (++) . fmap (:[])
```

deu pra entender a intenção mas tem type error assim!

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.

x : m (m c)

~~m a → (m a → m m a) → m m a~~

m a → (a → m m a) → m m a

m m b → (m b → m b) → m b

join x = bind x id

(66) **F**

bind x f :

(pure f) x

(12) **F1** Defina as typeclasses Functor e Applicative, e **enuncie** as leis de Functor.

Functor f: fmap :: (a→b) → f a → f b

Applicative f (supondo f functor): pure :: a → f a, (<*>) :: f (a→b) → f a → f b

Leis de functor: fmap id = id, fmap (f · g) = fmap f · fmap g

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join.

```
bind :: m a -> (a -> m b) -> m b        join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.
RESPOSTA. Não podes usar notação-do.

join x = bind x id

bind x f = join $ pure f x

Pode usar tudo de list e nada de maybe

(24) **F3.** Instancie Maybe e List como Functors, Applicatives, e Monads[3].
Sobre o List: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE Maybe. 3 coisas

```
fmap _ Nothing = Nothing
fmap f (Just x) = Just (f x)
pure = Just
(Just f) <*> (Just x) = Just (f x)
_ <*> _ = Nothing
return = Just
join (Just (Just x)) = Just x
join _ = Nothing
```

SOBRE List. (Dadas as funções da U1.) 4 coisas

```
fmap = map
pure1 x = [x]   pure2 x = [x]   repeat x
fs <*>1 xs = zipWith ($) fs xs
fs <*>2 xs = [f x | f <- fs, x <- xs]
return x = [x]
join = concat
fs <*>2 xs = foldr (λf acc → ys f ++ acc) [] fs
    where ys f = foldr (λx acc → f x : acc) [] xs
```

(6) **F4.** ... e descreva **curtamente** a *interpretação computacional* de cada.[4] (listas)

Inst 2: modela não-determinismo, cada elemento da lista é um possível valor

Inst 1: modela programação multi-thread: o i-ésimo elemento ~~de uma lista é um~~ de uma lista é um elemento da i-ésima thread do programa (ou processo)

(12) **F5.** Defina a *composição Kleisli* (denotada por (>=>) em Haskell), com tipo:

$$(>=>) : \text{Monad } m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

```
f >=> g = λx → join $ return g <*> f x
```

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

f >=> g x y ← f x

x : α

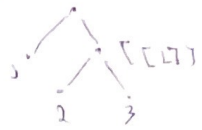
f x : m β

~~f >=> g~~ ~~return f x~~ : m (m β)

return g : m (β → m γ)

return g <*> f x : m m γ

fmap :: (a → b) → Tree a → Tree b



(54) # T

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R                          type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) fmap : ? ~~preorder~~ inorder fold : (α → α → α) → Tree α → α

(4×) forks, depth, tips : Tree α → Nat flat : Tree α → List α

(6×) search : α → Tree α → List Path fetch : Path → Tree α → Maybe α

quantos pontinhos

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap :: (a→b) → Tree a → Tree b
fmap f (Tip x) = Tip (f x)
fmap f (Fork t1 t2) = Fork u1 u2
  where u1 = fmap f t1
        u2 = fmap f t2

forks (Tip x) = Zero
forks (Fork t1 t2) = Succ (f1 + f2)
  where f1 = forks t1
        f2 = forks t2

tips (Tip x) = Succ Zero
tips (Fork t1 t2) = tips t1 + tips t2

depth (Tip x) = Succ Zero
depth (Fork t1 t2) = Succ (max d1 d2)
  where d1 = depth t1
        d2 = depth t2

fold _ (Tip x) = x
fold f (Fork t1 t2) = f (fold f t1) (fold f t2)
```

```
flat (Tip x) = [x]
flat (Fork t1 t2) = flat t1 ++ flat t2

fetch [] (Tip x) = Just x
fetch [] _ = Nothing
fetch _ (Tip x) = Nothing
fetch (x:xs) (Fork t1 t2) = if x == L then
  fetch xs t1 else fetch xs t2

search x (Tip y) = if x == y
    then [[]] else []
search x (Fork t1 t2) =
    map (L:) (search x t1) ++
    map (R:) (search x t2)
```

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições **devem** começar com o nome da função, seguido por um '='. Podes usar where e λambdas se quiser — mas *sem trollagens!*[5]

```
tips  = fold (+) . fmap (const 1)
depth = fold (λa,b → 1 + max a b) . fmap (const 0)

flat = fold (++) . fmap (λx → [x])
```

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.

depthize

fmap (λa,b → 1 + max a b)

(66) **F**

(12) **F1.** Defina as typeclasses `Functor` e `Applicative`, e **enuncie** as leis de `Functor`.

```
instance Functor (T a) where
  fmap :: (a -> b) -> (T a) -> (T b)
instance Applicative (T a) where
  ap :: (a -> b -> c) -> (T a) -> (T b) -> (T c)
```

Leis de Functor:
`fmap id = id`
`fmap (g . f) = fmap g . fmap f`

pure

(12) **F2.** Um applicative `m :: * -> *` consegue ser um Monad definindo uma bind ou uma join:

```
bind :: m a -> (a -> m b) -> m b        join :: m (m a) -> m a
```

Mostre que as duas abordagens são equivalentes, definindo cada uma em termos da outra.
RESPOSTA. Não podes usar notação-do.

(24) **F3.** Instancie Maybe e List como Functors, Applicatives, e Monads[3].
Sobre o List: dê **duas** maneiras diferentes de instanciar como Applicatives...

SOBRE Maybe.

```
instance Functor (Maybe a) where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just $ f x
instance Applicative (Maybe a) where
  pf <*> (Just x) = Just (pf x)
  _ <*> _ = Nothing
```

SOBRE List. (Dadas as funções da U1.)

```
instance Functor [a] where
  fmap = map
instance Applicative [a] where
  pf <*> xs = zipWith ($) pf xs
instance Applicative [a] where
  pf <*> xs = zipWith ($) pf xs
```

(6) **F4.** ...e descreva **curtamente** a *interpretação computacional* de cada.[4]

A #1 toma como prioridade a primeira lista e a #2 a segunda. É semelhante com a diferença entre o foldl e foldl1.

(12) **F5.** Defina a *composição Kleisli* (denotada por (>=>) em Haskell), com tipo:

$$(>=>) : \text{Monad } m \Rightarrow (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma)$$

RESPOSTA. (Podes usar notação-do, mas nesse caso a questão vale metade dos pontos.)

```
(>=>) f g x = do
  y <- f (g x)
  return y
```

x : α
g : β → mα
type error

[3] Podes usar tanto a definição baseada no `bind` quanto no `join`.
[4] Deixe claro qual é qual!

mesmo trocando f ↔ g
continua com type error:

x : α
g : α → mβ
f : β → mγ
⊢ gx : mβ
e f(gx) type error!

T

```
data Tree α = Tip α | Fork (Tree α) (Tree α)
data Step   = L | R
                                        type Path = [Step]
```

(36) **T1.** Levando em consideração os exemplos de uso no quadro, defina recursivamente as funções:

(4×) $\mathsf{fmap} : ?$ $\qquad\qquad$ $\mathsf{fold} : (\alpha \to \alpha \to \alpha) \to \mathsf{Tree}\ \alpha \to \alpha$

(4×) $\mathsf{forks}, \mathsf{depth}, \mathsf{tips} : \mathsf{Tree}\ \alpha \to \mathsf{Nat}$ $\qquad$ $\mathsf{flat} : \mathsf{Tree}\ \alpha \to \mathsf{List}\ \alpha$

(6×) $\mathsf{search} : \alpha \to \mathsf{Tree}\ \alpha \to \mathsf{List\ Path}$ $\qquad$ $\mathsf{fetch} : \mathsf{Path} \to \mathsf{Tree}\ \alpha \to \mathsf{Maybe}\ \alpha$

RESPOSTA. (**Não** repita as tipagens na resposta! (Ou seja, escreva apenas da fmap.))

```
fmap : (a → b) → Tree a → Tree b
fmap f (Tip x) = Tip (f x)
fmap f (Fork l r) = Fork l' r'
  where
  l' = fmap f l
  r' = fmap f r

fold _ (Tip x) = x
fold f (Fork l r) = l' `f` r'
  where
  l' = fold f l
  r' = fold f r

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

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

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

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

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

(18) **T2.** Mostre como definir as tips, depth, flat como composições, em estilo point-free, usando ambas as fold, fmap para cada uma delas. Todas as definições devem começar com o nome da função, seguido por um '='. Podes usar where e λambdas se quiser — mas *sem trollagens!*[5]

```
tips  = fold (+) . fmap (\x → 1)
forks = (-1) . tips
flat  = fold (++) . fmap (\x → [x])
```

Só isso mesmo.

[5] DEFINIÇÃO. Uma resposta é *trollagem* sse Thanos a acha trollagem.