(12) **A**

12

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

~~Bool~~ `data Bool = True | False`
`data Nat = zero | Succ Nat`
`data List a = Empty | Cons a (List a)`
`data Maybe a = Nothing | Just a`

(16) **B**

16

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

p :: α → Bool
f :: α → β

Infira seu tipo

`pick` :: `(α → Bool) → (α → β) → [α] → [β]`

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

`pick p f =` `map f . filter p`

(β). α↦ map f : [α] → [β]

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

12
```
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

12
```
zipWith f xs ys = map (λ(z,z') → f z z') zs
  where zs = zip xs ys
```
(uncurry f)

```
pairs = map (λx → (x,x))
```
↳ pairs [1,2,3,4] = [(1,2),(2,3),(3,4)]

12
```
takeFirst _ [] = Nothing
takeFirst p (x:xs) = if p x then Just x else takeFirst p xs
```

12
```
inits [] = [[]]
inits (x:xs) = [] : map (x:) (inits xs)
```

é melhor definir a zip usando a zipWith (como?) do que o contrário

12
```
enumFrom x = x : enumFrom (succ x)
```

AUXILIAR ✓
```
map _ [] = []
map f (x:xs) = f x : map f xs
```

Só isso mesmo.

(12) A

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

```
data Bool = True | False
data Nat = zero | succ Nat
data List = List | List a
```

(16) B

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

`pick` :: `(α -> Bool) -> (α -> β) -> [α] -> [β]`

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

`pick p f =` `filter f . filter p` ~~filter~~ map

(72) C

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

```
zip _ [] = []
zip [] _ = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

enumFrom x = x : enumFrom (succ x)

zipWith f _ [] = []
zipWith f [] _ = []
zipWith f (x:xs) (y:ys) = (x,y) : zipWith f xs ys
map f [] = []
map f (x:xs) = | f x = x : filter f xs
               | otherwise = filter f xs

takeWhile f (x:xs) = | f x = x : takeWhile f xs
                     | otherwise = []
```

Só isso mesmo.

8 (12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

3 `data Bool = True | False`	2 `data List = L a (List a) | Empty`
7 `data Nat = Zero | Succ Zero`	⋮

↳ Não deu tempo de estudar Maths

14 (16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Just ⁝

Infira seu tipo

6 `pick` :: `(a → Bool) → (a → b) → [a] → [b]`

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

8 `pick p f =` `map f . filter p`

6/ (72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

12
```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _ = []
```

12
```
zipWith f (x:xs) (y:ys) = f a b : zipWith xs ys
```

11
```
inits [] = [[]]
inits l@(_:xs) = l ++ inits xs
```

6
```
Perms [a] = "Enough"   ← idéia ruim
Perms [] = []
Perms (x : xs@(x' : xs')) = (x,x') : Perms xs
```
→ error "foo"
assim só funcionaria corretamente para listas infinitas. (por quê?)

8
```
(++) [] xs = xs
(++) xs [] = xs
(++) (x:xs) ys = x : (xs ++ ys)
```

Só isso mesmo.

12
```
enumFrom x = [x] ++ enumFrom (succ x)
```
X ⁝

10

(12) A

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

3 3 3 1

```
data Bool = True | False
data Nat = Zero | Succ Nat
data List a = [] | a : List a
data Maybe a = Nothing | a
```

Just

16

(16) B

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

8 `pick ::` `(a -> Bool) -> (a -> b) -> [a] -> [b]`

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

8 `pick p f =` `map f . filter p`

(72) C

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

12 (1 linha)

```
zip _ [] = []
zip [] _ = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

8

```
takeFirst f [] = Nothing
takeFirst f (x:xs) | f x = x | otherwise = takeFirst f xs
```

Just x

12 (1 linha)

```
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

11

```
enumFrom x = x : enumFrom (x+1)
```

(Succ x)

!! 6 (outra função)

```
pairs [] = []
pairs (x:xs) = (x,x) : pairs xs
```

~~inits [] = [[]]~~

~~inits (x:xs) = (x:xs) :~~

inits xs = xs : inits Só isso mesmo.

(init xs)

12

```
where
  init [] = []
  init [x] = []
  init (x:xs) = x : init xs
```

(1 linha)

(12) **A**

Defina os tipos de dados:

12

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

```
data Bool = False | True        data List a = Nil | Cons a (List a)
data Nat = zero | Succ Nat      data Maybe a = Nothing | Just a
```

(16) **B**

16

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

`pick` :: `(a → Bool) → (a → b) → [a] → [b]`

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

`pick p f` = `(map f) . (filter p)`

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

```
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

takeFirst _ [] = Nothing
takeFirst p (x:xs)
  | p x = Just x
  | otherwise = takeFirst p xs

zipWith f = (uncurry f) . zip
  where
    uncurry f (x,y) = f x y

pairs (x:x':xs) = (x,x') : pairs (x':xs)
pairs _ = []
inits [] = [[]]
inits (x:xs) = [] : map (x:) (inits xs)
  where
    map _ [] = []
    map f (x:xs) = f x : map f xs

enumFrom n = n : enumFrom (Succ n)
```

12 12 12 12 10 12

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

```
data Bool = False | True        data List a = Empty | Cons (List a)
data Nat = Zero                 data Maybe a = Nothing
         | Succ Nat                          | Just a
```

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

`pick ::` `(a -> Bool) -> (a -> b) -> [a] -> [b]`

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

`pick p f =` `map f . filter p`

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

```
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

imito [] = [[]]
imito (x:xs) = (x:xs) : imito xs

takeFirst _ [] = Nothing
takeFirst p (x:xs)
    | p x = Just x
    | otherwise = takeFirst p xs

enumFrom n = n
    n : enumFrom (Succ n)

zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys

pairs [] = []
pairs [x] = []
pairs (x1:x2:xs) = (x1,x2) : pairs (x2:xs)
```

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

```
data List = Empty | Cons Int
data Nat = Zero | Succ
data Bool = True | False
```

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

```
pick :: (a -> Bool) -> (a -> b) -> [a] -> [b]
```

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

```
pick p f = (filter p) $ (map f)
```

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

```
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys

zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

pairs [] = []
pairs [x] = (x,[])
pairs (x:y:zs) = (x,y) : pairs zs

enumFrom x = x : enumFrom (Succ x)

map _ [] = []
map f (x:xs) = f x : map f xs

takeWhile _ [] = []
takeWhile f (x:xs)
  | f x = x : takewhile f xs
  | otherwise = []
```

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

12

RESPOSTA.

```
DATA BOOL = TRUE | FALSE
DATA NAT = ZERO | SUCC NAT
DATA LIST A = EMPTY | CONS A (LIST A)
MAYBE A = NOTHING | JUST A
```

(16) **B**

16

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

`pick` :: $(A \to BOOL) \to (A \to B) \to [A] \to [B]$

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

```
pick p f = (MAP F) . (FILTER P)
```

69 (72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

12
```
ZIP _ [] = []
ZIP [] _ = []
ZIP (X:XS) (Y:YS) = (X,Y) : ZIP XS YS
```

12
```
ZIPWITH _ [] _ = []
ZIPWITH _ _ [] = []
ZIPWITH F (X:XS) (Y:YS) = F X Y : ZIPWITH F XS YS
```

12
```
TAKEFIRST _ [] = NOTHING
TAKEFIRST P (X:XS) = IF (P X) THEN JUST X (ELSE TAKEFIRST P XS)
```

10
```
PAIRS [] = []
PAIRS [_] = []
PAIRS (X:Y:XS) = (X,Y) : PAIRS XS
```

12
```
ENUMFROM N = N : ENUMFROM (SUCC N)
```

11
```
INITS [] = [] (NÃO PODE)
INITS X = X : INITS (INIT X)
```

AUXILIAR: INIT
```
INIT [] = []
INIT [X] = []
INIT (X:XS) = X : INIT XS
```

Só isso mesmo.

(12) **A**

No ghci veja o :kind

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

6

RESPOSTA: bugou na sintaxe

```
Bool :: True        Nat :: Zero        List :: a→[a]
Bool :: False       Nat :: Succ Nat    ?
```

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

8 `pick` :: `(a→Bool)→(a→b)→[a]→[b]`

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

8 `pick p f` = `map f . filter p`

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

8
```
map :: (a→b)→[a]→[b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

8
```
filter :: (a→Bool)→[a]→[a]
filter _ [] = []
filter p (x:xs)
  | p x = x : filter p xs
  | otherwise = filter p xs
```

8
```
(.) :: (b→c)→(a→b)→a→c
(.) f g x = f (g x)
```

12
```
zip :: [a]→[b]→[(a,b)]
zip _ [] = []
zip [] _ = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

12
```
zipWith :: (a→b→c)→[a]→[b]→[c]
zipWith _ _ [] = []
zipWith _ [] _ = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

12
```
enumFrom :: Nat → [Nat]
enumFrom x = x : enumFrom (Succ x)
```

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

```
Bool :: *        Nat :: *        List :: * → *        Maybe :: * → *
```

RESPOSTA.

3 `data Bool = True | False`

2 `data Nat = Zero | Succ a`

2 `data List a = Empty | Cons a (List a)` 3

2 `data Maybe a = None | Just a`

donde chegou?

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

6 `pick :: (a -> Bool) -> (a -> b) -> List a -> List b`

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

0 `pick p f = [f x | x <- [0..], p x]`

?!

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

8
```
map _ [] = []
map f (x:xs) = f x : map f xs
```

12
```
takeFirst _ [] = None
takeFirst p (x:xs) = if p x then Just x
                     else takeFirst p xs
```

6
```
drop _ [] = []
drop 0 xs = xs
drop (n+1) (x:xs) = drop n xs
```
Nat!

12
```
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

8
```
filter _ [] = []
filter p (x:xs) = if p x then x : filter p xs
                  else filter p xs
```

12
```
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = (f x y) : zipWith f xs ys
```

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

```
Data Bool = True | False
Data Nat = Zero | Succ Nat
Data List a = [] | (:) a (List a)
```

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

0 `pick` :: $(a \to b) \to (b \to c) \to (List\ c)$

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

0 `pick p f =` `f _ : pick p _`

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

```
map f [] = []
map f (x:xs) = x : map f xs

(++) [] ys = ys
(++) (x:xs) ys = x : (xs ++ ys)

drop _ [] = []
drop Zero xs = xs
drop (S n) (x:xs) = drop n xs

enumFrom n = n : enumFrom (S n)

(.) g f x = (g . f) x
```

```
Pair [] = []
Pair [x] = Pair []
Pair (x:y:xs) = (x,y) : Pair (y:xs)
```

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

```
Bool :: *        Nat :: *        List :: * → *        Maybe :: * → *
```

RESPOSTA.

12

```
data Bool = True | False
data Nat = Zero | Succ Nat
data List a = Empty | Cons a (List a)
data Maybe a = Nothing | Just a
```

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

8

```
pick :: (a → Bool) → (a → b) → [a] → [b]
```

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

8

```
pick p f = map f . filter p
```

(72) **C**

6?

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

12

```
zip _ [] = []
zip [] _ = []
zip (x:xs) (y:ys) = (x,y) : (zip xs ys)
```

7

```
map _ [] = []
map f (x:xs) = f x : (map f xs)
```

12

```
takeFirst _ [] = Nothing
takeFirst p (x:xs)
  | p x = Just x
  | otherwise = takeFirst xs
```

8

```
filter _ [] = []
filter p (x:xs)
  | p x = x : filter xs
  | otherwise = filter xs
```

12

```
zipWith _ [] = []
zipWith _ [] _ = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

12

```
enumFrom n = n : enumFrom (Succ n)
```

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

3 (3?)

```
data Bool = False | True .   data List a = Nil | Cons a (List a)
data Nat = Zero | Succ Zero .   data Maybe a = Nothing | Just a .
```

cadê a recursão?

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

8 `pick ::` `(a -> Bool) -> (a -> b) -> [a] -> [b]`

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

5 `pick p f =` `Filter p . Map f`

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

(1.) 8
```
map F (x:xs) = F x : map F xs
map _ [] = [].
```

(2). 8
```
Filter P (x:xs) = if P x then
  x : Filter xs else Filter xs .
Filter _ [] = [].
```

(3) 8
```
drop (Succ n) (x:xs) = drop n xs
drop Zero xs = xs.
```

(4) 12
```
Zipwith F (x:xs) (y:ys) = (F x y) : Zipwith F xs ys
Zipwith F [] ys = []
Zipwith F xs [] = []
```

(5) 12
```
Zip xs ys = Zipwith (,) xs ys
```

(6) 8
```
Replicate (Succ n) a
  = a : Replicate n a
Replicate Zero _ = []
```

Só isso mesmo.

(12) **A**

12 Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

3 `data Bool = True | False` 3 `data List a = Empty | Cons a (List a)`

7 `data Nat = Zero | Succ Nat` `data Maybe a = Nothing | Just a` 3

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

8 `pick` :: `(a -> Bool) -> (a -> b) -> [a] -> [b]`

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

8 `pick p f =` `map f . filter p`

(72) **C**

65

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

12
```
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
```

10
```
pairs [] = []
pairs [x] = []
pairs (x:y:xs) = (x,y):(pairs xs)
```

12
```
takeFirst _ [] = Nothing
takeFirst p (x:xs)
    | p x = Just x
    | otherwise = takeFirst p xs
```

11
```
inits [] = []
inits xs = xs:(inits (init xs))
  where
    init [] = []
    init [y] = []
    init (y:ys) = y:(init ys)
```

12
```
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = (f x y):(zipWith f xs ys)
```

8
```
map _ [] = []
map f (x:xs) = (f x):(map f xs)
```

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

0 RESPOSTA.

Bool :: Bool
NAT :: NAT
LIST :: a → LISTa
MAYBE :: a → MAYBE a

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

2 `pick` :: (b → c) → (a → b) → List a → List b

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

0 `pick p f =` P (F xs)

donde chegou?

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

```
8 MAP :: (a → b) → List a → List b
  MAP _ [] = []
  MAP f (x:xs) = f x : MAP xs

3 (+) :: Nat → Nat → Nat
  (+) ZERO x = x
  (+) x (SUCC y) = SUCC ((+) x y)

4 (&&) :: Bool → Bool → Bool
  (&&) TRUE TRUE = TRUE
  (&&) _ _ = FALSE

4 len :: List a → NAT
  len [] = ZERO
  len (x:xs) = SUCC (len xs)

  ALL :: (a → Bool) → List a → Bool
  ALL _ [] = FALSE TRUE!
  ALL p (x:xs) =
    CASE p x == TRUE OF
      TRUE → CASE LEN XS != ZERO OF
               TRUE → ALL XS
               FALSE → P X
      FALSE → P X → por quê?

7 FILTER :: (a → Bool) → List a → List a
  FILTER _ [] = []
  FILTER P (x:xs) =
    CASE P X == TRUE OF
      TRUE → X : FILTER XS
      FALSE → FILTER XS
```

nunca!!

não faz sentido contar o tamanho duma lista simplesmente para decidir se ela é vazia!

isso não parece programação funcional

Só isso mesmo.

(12) **A** data hist = Emptyhist |

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

3 data Bool = True | False data hist a = Empty X | ?

7 data Nat = Zero | Succ Nat

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

7 `pick` :: $(\alpha \to Bool) \to (\alpha \to \beta) \to [\alpha] \to [\beta]$

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

8 `pick p f =` `(.) (map f) (filter p)`

map f · filter p
mais legível

(72) **C**

(:)

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

(1) 12
```
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
```

(2) 12
```
inits [] = [[]]
inits (x:xs) = (x:xs):inits xs
```

(3) 12
```
zipwith f _ [] = []
zipwith f [] _ = []
zipwith f (x:xs) (y:ys) = (f x y):(zipwith xs ys)
```

12
```
enumFrom x = x:enumFrom (succ x)
```
(4)

11
```
pairs [] = error "lista vazia"
pairs [_] = []
pairs (x:xs) = (x, head xs):pairs xs
(x1:x2:xs) = ...
where
  head (x:xs) = x
  head [] = []
```
por que não pattern-match?
(5)

8
```
map f [] = []
map f (x:xs) = (f x):map f xs
```
(6)

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

instance Bool True | False 3 instance List [] | (x : xs) ? ?	instance Nat 0 | Sn 2 ?

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

6 `pick` :: $(a \to Bool) \to (a \to b) \to [a] \to [b]$

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

`pick p f` = `f x : pick p f x`

donde chegam?

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

8
```
map f [] = []
map f (x:xs) = f x : map f xs
```

~~repeat 0 = [] 7~~
```
repeat n = n : repeat n
```

12
```
inits [] = [[]]
inits (x:xs) = [] : map (:x) : inits xs
```

8
```
replicate 0 x = []
replicate Sn x = x : replicate n x
```

```
enumFrom n =
    n : enumFrom Sn
```
12

(8)
```
drop 0 [] = []
drop 0 xs = xs
drop (Sn) (x:xs) = drop n xs
```

drop 2 [] = ?

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

0

Bool :: Integer

Nat :: Integer

List :: Integer → [] or Character → []

Maybe :: Integer → Bool or Character → Bool

(16) **B**

0

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

`pick` :: Nat → Nat → Nat → Nat

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

`pick p f` = f(p(x))

(72) **C**

0

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

!

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

3 Data Bool = true | False 3 Data Nat = Zero | Succ Nat
Data List a = [] | [a] 3 Data Maybe a = Just a | Nothing

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

4 `pick` :: (a → Bool) → (a → a) → [a] → [a]

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

0 `pick p f` = [P x]

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

&&

4 true && true = true
— && — = False

4 Even Zero = true
Even (Succ Zero) = False
Even (Succ (Succ m)) = Even m

4 m + Zero = m
m + (Succ m) = Succ (m + m)

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

```
data Bool = True | False
data Nat = Zero | Succ Nat
data List a = Empty | Cons a (List a)
data Maybe a = Nothing | Just a
```

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

`pick` :: `(a → Bool) → (a → b) → [a] → [b]`

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

`pick p f` = `map f . filter p`

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

```
zip _ [] = []
zip [] _ = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

takeFirst _ [] = Nothing
takeFirst p (x:xs)
  | p x = Just x
  | otherwise = takeFirst p xs

enumFrom n = n : enumFrom (succ n)
```

```
zipWith _ _ [] = []
zipWith _ [] _ = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys

pairs [] = []
pairs [x] = []
pairs (x:y:xs) = (x,y) : pairs (y:xs)

inits [] = [[]]
inits xs = xs : inits (init xs)
  where
    init [] = error "Empty List"
    init [y] = []
    init (y:ys) = y : init ys
```

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

3 `data Bool = True | False`

3 `data Nat = ZERO | SUCC Nat`

2 `data List x = EMPTY | List x (List x)`

3 `data Maybe a = Nothing | Just a`

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

8 `pick` :: `(a → Bool) → (a → b) → [a] → [b]`

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

8 `pick p f` = `map f . Filter p`

(72) **C**

66

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

9
```
TakeFirst _ [] = []   Nothing
TakeFirst p (x:xs)
  | p x = Just x
  | otherwise = TakeFirst p xs
```

12
```
Pairs [] = []
Pairs [x] = []
Pairs (x1:x2:xs) = (x1,x2) : Pairs (x2:xs)
```

12
```
EnumFrom x = x : EnumFrom (Succ x)
```

12
```
Zip [] _ = []
Zip _ [] = []
Zip (x:xs) (y:ys) = (x,y) : Zip xs ys
```

9
```
Inits [] = [[]]
Inits xs = xs : Inits $ Take (Len xs) - 1) xs
  where
    Take _ [] = []
    Take 0 _ = []
    Take I (x:xs) = x : Take (I-1) xs
```

Nat

ideia ruim. (Por quê?)

12
```
ZipWith _ [] _ = []
ZipWith _ _ [] = []
ZipWith f (x:xs) (y:ys) = (f x y) : ZipWith f xs ys
```

Só isso mesmo.

Int

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA. bugou na notação

```
Bool :: True | False
Nat :: zero | succ Nat
List :: a → [a]
Maybe :: a → Nothing | Just a
```

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

`pick` :: `(a → Bool) → (a → B) → ? → ?`

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

```
pick p f =
```

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

```
map f [] = []
map f (x:xs) = f x : map f xs

filter p [] = []
filter p (x:xs)
  | p x = x : filter xs
  | otherwise = filter p xs

takefirst p [] = Nothing
takefirst p (x:xs)
  | p x = Just x
  | otherwise = takefirst p xs

zip [][] = []
zip (x:_)[] = [x]
zip [](y:_) = [y]
zip (x:xs)(y:ys) = (x,y) : zip xs ys

(++) xs [] = xs
(++) [] ys = ys
(++) xs ys = xs : ys

drop _ [] = []
drop zero xs = xs
drop (succ n) xs = drop n xs
```

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

12

```
data Bool = False | True
data Nat = Zero | Succ Nat
data Maybe a = Nothing | Just a
data List a = EmptyList | Cons a (List a)
```

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

8 `pick ::` `(a -> Bool) -> (a -> b) -> [a] -> [b]`

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

8 `pick p f =` `map f . filter p`

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

12
```
takeFirst _ [] = Nothing
takeFirst p (x:xs) = if p x then Just x else takeFirst p xs
```

12
```
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

12
```
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

12
```
pairs [] = []
pairs [_] = []
pairs (x:x':xs) = (x,x') : pairs (x':xs)
```

11
```
inits [] = [[]]
inits (x:xs) = [x:xs] : inits xs
```
type error

Só isso mesmo.

10
```
enumFrom n = [n] ++ enumFrom (succ n)
  where (++) [] xs = xs
        (++) xs [] = xs
        (++) (x:xs) ys = x : (xs ++ ys)
```
n: !!

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

9

~~data Bool where TRUE \| FALSE~~ data Bool where TRUE \| FALSE
~~data Nat where~~ data Nat where ZERO \| Succ ZERO
data List where Empty \| Conn a (List a)

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

8 `pick ::` ~~(a → Bool) → (a → b) → List a →~~ (a → Bool) → (a → b) → List a → List b

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

2 `pick p f =` pick' xs = map f (filter p xs) xs

?!?

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

cadê a recursão?!

```
map _ [] = []
map f (x:xs) = f x : xs
```

5

```
(&&) _ FALSE = FALSE
(&&) FALSE _ = FALSE
(&&) _ _ = TRUE
```

descomsiderar

```
(+) x ZERO = x
(+) x (Succ y) = Succ ((+) x y)
```

```
drop _ [] = []
drop ZERO (x:xs) = (x:xs)
drop (Succ n) (x:xs) = drop n xs
```

(8)

por quê?

```
all _ [] = FALSE
all f (x:[]) = f x
all f (x:xs) = (&&) (f x) (all f xs)
```

5 — necessário?

que tal escrever infixo?

```
replicate ZERO _ = []
replicate (Succ n) x = x : replicate n x
```

8

```
repeat x = x : repeat x
```

8

```
filter _ [] = []
filter f (x:xs) = if (f x) then x :
filter f xs else filter f xs
```

8

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

```
data Bool = True | False
data Nat = Zero | Succ Nat
data List a = Empty | Cons a (List a)
data Maybe a = Just a | Nothing
```

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

```
pick :: (a → Bool) → (a → b) → List a → List b
```

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

```
pick p f = (:) p 0
```

(72) **C**

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

```
(&&) :: Bool → Bool → Bool
(&&) True True = True
(&&) True False = False
(&&) False _ = False

all :: (a → Bool) → List a → Bool
all (True (List a)) = True
all (False (List a)) = False

even :: Nat → Bool
even x = True
even _ = False
  where x `mod` 2 == 0

len :: List a → Nat
len [] = Empty 0
len [x] = 1
len (x : xs) = 1 + (len xs)

replicate :: Nat → a → List a
replicate 0 _ = 0
replicate 1 y = y
replicate x y = y ++ (replicate (x-1) y)
```

precisaria definir

Só isso mesmo.

(12) **A**

Defina os tipos de dados:

`Bool :: *` `Nat :: *` `List :: * → *` `Maybe :: * → *`

RESPOSTA.

3 3 3

```
data Bool = True | False
data Nat = Zero | Succ Nat
data List a = Empty | Cons a (List a)
```

(16) **B**

Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

8 `pick` :: $(a \to Bool) \to (a \to b) \to [a] \to [b]$

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

0 `pick p f` ~~`(x:xs)`~~ ~~`if (p x) then (f x):(pick p f xs) else []`~~ ~~`(pick p f xs)`~~

(72) **C**

54

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

12 ① 8 ⑤
```
zip _ [] = []
zip [] _ = []
zip (x:xs) (y:ys) = (x,y):(zip xs ys)

len [] = Zero
len (_:xs) = Succ len xs
```

12 ② 8 ⑥
```
zipWith _ _ [] = []
zipWith _ [] _ = []
zipWith f (x:xs) (y:ys) = (f x y):(zipWith f xs ys)

all p [] = True
all p (x:xs) = (p x) && (all p xs)
```

6 ③
```
map f [] = []
map f (x:xs) = (f x):(map f xs)
```

8 ④
```
filter p [] = []
filter p (x:xs) = if (p x) then x:(filter p xs) else (filter p xs)
```

Só isso mesmo.

(12) **A**

6 Defina os tipos de dados:

```
Bool :: *        Nat :: *        List :: * → *        Maybe :: * → *
```

RESPOSTA.

DATA Bool where | TRUE | FALSE	DATA NAT where | ZERO | SUCC NAT	DATA List where | Empty = [ ] | La a = [a]	
4	1	1	

(16) **B**

0 Considere a função `pick` definida assim, usando list comprehension:

```
pick p f xs = [f x | x ← xs, p x]
```

Infira seu tipo

0 `pick` :: $(a \to a) \to Bool \to [a] \to (a \to a)$

e mostre como ela pode ser definida numa linha só, sem usar lambdas, começando assim:

0 `pick p f (x:xs) =` $fx : pick\ p\ f\ x_0$

(72) **C**

34 (++) ≠ (:) !!!

Escolha **até 6** das funções da primeira página par definir. É **proibido** usar list comprehension. Veja *bem* os tipos, pois podem ser diferentes dos escolhidos pelo Prelude da Haskell.

DEFINIÇÕES.

```
(++) :: [a] → [a] → [a]
(++) x0 y0 = x0 : y0
```

```
h :: [a] → a
h (x:x0) = x
```

```
pairs :: [a] → [(a,a)]
pairs [] = []
pairs (x:x0) = (x, h x0) : pairs x0
```

```
zip :: [a] → [b] → [(a,b)]
zip [] _ = []
zip _ [] = []
zip (x:x0) (y:y0) = (x,y) : zip x0 y0
```

```
enumFrom :: Int → [Int]
enumFrom x = x : enumFrom (succ x)
```

```
zipWith f (x:x0) (y:y0) = f x y : zipWith f x0 y0
```

```
takeFirst x:x0 = a ++ takeFirst x0
  where a = x
        if f x == True then x
        otherwise = []
```

```
takeTo [] = []
takeTo (x:x0) = x : takeTo x0
```

type error

se xs = []?

NUNCA!

10

10

2

6

0X

6

Só isso mesmo.

deveria usar pattern-matching pra isso