(+1 1 [ ]: / (cons w)

(35) **A** Com tipos.

Defina os: Nat, List, map, filter, fold, curry, uncurry.

DEFINIÇÕES.

```
Nat :: Type
data Nat = 0 | S n

List :: Type → Type
data List α = Nil | Cons α (List α)

uncurry :: α → β → (α, β)
uncurry x y = (x, y)

map :: (α → β) → List α → List β
map _ [] = []
map f (x:xs) = (f x) : map f xs

filter :: (α → Bool) → List α → List α
filter _ [] = []
filter p (x:xs) | p x = x : (filter p xs)
                | otherwise = filter p xs

fold :: (α → α → α) → α → List α → α
fold _ i [] = i
fold f i (x:xs) = f x (fold f i xs)

curry :: (α, β) → α → β
curry (x, y) = y . x
```

quem é? ✓ ✓ ✓ ✓ ✗ ✗

(32) **B** Sem tipos.

Escolha **até 4** 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.

```
Escolhidas: zip, zipWith, countdown, pairs

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

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

pairs (a:b:l) = (a,b) : (pairs (b:l))
pairs _ = []

countdown 0 = [0]
countdown (S n) = (S n) : countdown n
```

✓ ✓ ✓ ✓

(12) **C**

Defina zip em termos da zipWith e zipWith em termos da zip.

```
zip = zipWith (\x -> (\y -> (x, y)))
zipWith f xs = (map (\(x,y) -> f x y)) . zip xs
```

(,) uncurry f

(21) **D**

Considere o tipo `Int` dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados `ArEx` para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias $(+)$ e $(\cdot)$ e a operação unária $(-_)$.

```
data Op =
  | BinOp :: (Int -> Int -> Int) -> Op
  | UnOp :: (Int -> Int) -> Op

data ArEx =
  | BinEx :: (ArEx -> ArEx -> BinOp) -> ArEx
  | UnEx :: (ArEx -> UnOp) -> ArEx
  | IntEx :: Int -> ArEx
```

↑ assim vai permitir arbitrárias ops (teu tipo vai ter lixo)

(complicou demais)

(7) **D2.** Defina uma função `eval` de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

```
eval :: ArEx -> Int
eval (IntEx n) = n ✓
eval (UnEx x op) = op (eval x) ✓
eval (BinEx lx rx op) = op (eval lx) (eval rx) ✓
```

(7) **D3.** Defina uma função `height` que retorna a altura da árvore sintáctica da sua entrada.

```
height :: ArEx -> Int
height (IntEx n) = 0
height (UnEx x _) = height x + 1
height (BinEx lx rx _) = (max (height lx) (height rx)) + 1 ✓
```

auxiliar:

```
max :: Int -> Int -> Int
max a b = if a>b then a else b
```

↓ ?

Só isso mesmo.

(35) **A**

Defina os: `Nat`, `List`, `map`, `filter`, `fold`, `curry`, `uncurry`.

DEFINIÇÕES.

```
Data Nat
  O : Nat
  S : Nat → Nat

Data List α
  Nil : List α
  Cons : α → List α → List α

Curry :: ((α, β) → γ) → (α → β → γ)
Curry (a, b) c = (a) b) c

Uncurry :: (α → β → γ) → ((α, β) → γ)
Uncurry a b c = (a, b) c

filter :: (α → Bool) → List α → List β
filter p [] = []
filter p (x::xs) = if Px then x::(filter p xs) else filter p xs
```

ev [1, 2]
ev 2
2 :: []

(32) **B**

Escolha **até 4** 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.

```
(+) n O = n
(+) n (S m) = S((+) n m)

(*) n (S O) = n
(*) n (S m) = (+) n (* n m)

Odd (S O) = True
odd O = False
odd S(S n) = odd n

Len [] = O
Len (x::xs) = S (len xs)
```

weird order

(12) **C**

Defina `zip` em termos da `zipWith` e `zipWith` em termos da `zip`.

```
Zipw f [] [] = []
Zipw f (x::xs) (y::ys) = f zip x::xs y::ys

Zip [n] [] = [n]
Zip (x::xs) (y::ys) = Zipw
```

type errors!

(21) **D**

Considere o tipo `Int` dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados `ArEx` para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias $(+)$ e $(\cdot)$ e a operação unária $(-_)$.

```
Typeclass ArEx Int
  (+) : Int -> Int -> Int
  (·) : Int -> Int -> Int
  (-_) : Int -> Int
```

(7) **D2.** Defina uma função `eval` de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

```
eval :: ArEx Int => Int
```

?

(7) **D3.** Defina uma função `height` que retorna a altura da árvore sintáctica da sua entrada.

```
height :: ArEx
```

Só isso mesmo.

(35) **A**

Defina os: Nat, List, map, filter, fold, curry, uncurry.

DEFINIÇÕES.

```
data Nat where
  O : Nat
  S : Nat → Nat

filter : (α -> Bool) -> List α -> List α
filter _ [] = []
filter p (x::xs) =
  | p x == True = x::(filter p xs)
  | o.w. = filter p xs

data List α where
  Nil : List α
  Cons : α -> List α -> List α

fold : (α -> α -> α) -> List α -> α -> α
fold _ [] i = i
fold f (x::xs) i = (f x) 'f' (fold f xs i)

map : (α -> β) -> List α -> List β
map _ [] = []
map f (x::xs) = f x::(map f xs)

curry : (α x β -> γ) -> α -> β -> γ
curry f a b = f(a,b)

uncurry : (α -> β -> γ) -> (α x β) -> γ
uncurry f (a,b) = f a b
```

NUNCA!!

por que enfatisar isso?!!?

type error

(32) **B**

Escolha **até 4** 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.

```
countDown : Nat -> List Nat
countDown O = [O]
countDown Sn = (Sn)::(countDown n)

zip : List α -> List β -> List(α x β)
zip (x::xs) (y::ys) = (x,y)::(zip xs ys)
zip _ _ = []

zipWith : (α -> β -> γ) -> List α -> List β -> List γ
zipWith f la lb = map (uncurry f) (zip la lb)

Pair : List α -> List (α,α)
Pair (x::(x'::xs)) = (x,x')::(Pair (x'::xs))
Pair _ = []
```

opcionais

necessárias

(12) **C**

Defina zip em termos da zipWith e zipWith em termos da zip.

B

(21) **D**

Considere o tipo `Int` dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados `ArEx` para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias $(+)$ e $(\cdot)$ e a operação unária $(-_)$.

```
data ArEx where
  Bop : (Int -> Int -> Int) -> ArEx -> ArEx -> ArEx
  Vop : (Int -> Int) -> ArEx -> ArEx
  Num : Int -> ArEx
  Nil : ArEx
```

(7) **D2.** Defina uma função `eval` de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

```
eval : ArEx -> Int
eval Nil = 0          (por quê?)
eval (Num n) = n
eval (Vop n) = Vop n
eval (Bop n m) = Bop n m
```

este construtor não era binário?
e este ternário?

(7) **D3.** Defina uma função `height` que retorna a altura da árvore sintáctica da sua entrada.

```
height : ArEx -> Nat
height Nil = 0
height (num _) = S0
height (Vop _) = S0
height (Bop l r) = S (max l r)
```

) x

```
max : ArEx -> ArEx -> ArEx
max a Nil = a
max Nil a = a
max (Num _) a = a
max a (Num _) = a
  ⋮
```

Putz, cabou no espaço ó

grrr...

Só isso mesmo.

(35) **A**

Defina os: `Nat`, `List`, `map`, `filter`, `fold`, `curry`, `uncurry`.

DEFINIÇÕES.

```
DATA NAT
  O : NAT
  S(n) : NAT → NAT

DATA LIST : α → LIST α → LIST α          ?
  NIL = [ ]                                ?
  CONS (X:XS) = X :: CONS XS               ✗

DATA MAP : (α → β) → L α → L β
  MAP F [ ] = [ ]
  MAP F (X:XS) = FX :: MAP F XS            ✓

FILTER : (α → BOOL) → L α → L α
  FILTER P [ ] = [ ]
  FILTER P (X:XS)
    | PX = X :: FILTER P XS
    | OTHERWISE = FILTER P XS              ✓

FOLD : α → (α × α → α) → LIST α → α
  FOLD R B [ ] = R
  FOLD R B (X:XS) = X 'B' (FOLD R B XS)    ✓

CURRY : (α × α → α) → (α → α → α)          β  δ ...
  F(A,B) = F A B                            ?

UNCURRY : (α → α → α) → (α × α → α)
  F A B = F(A,B)                            ✗
```

(32) **B**

Escolha **até 4** 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.

```
COUNTDOWN                      VALE 8     oops! [0] type error
  COUNTDOWN O = [O]
  COUNTDOWN S n = S n :: COUNTDOWN n

ZIP                            VALE 8     ✗
  ZIP [ ] _ = [ ]
  ZIP _ [ ] = [ ]
  ZIP (X:XS) (Y:YS) = (X,Y) :: ZIP (XS) (YS)

INITS                          VALE 8     ✓
  INIT [ ] = [[ ]]
  INIT (X:XS) = [X] :: INIT XS            ✗

VALE 4
ODD
  ODD O = FALSE
  ODD (S n) = EVEN n
EVEN
  EVEN O = TRUE
  EVEN (S n) = ODD n
```

(12) **C**

Defina `zip` em termos da `zipWith` e `zipWith` em termos da `zip`.

(21) **D**

Considere o tipo `Int` dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados `ArEx` para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias $(+)$ e $(\cdot)$ e a operação unária $(-_)$.

ArEx :

(7) **D2.** Defina uma função `eval` de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

EVAL : ArEx → INT

(7) **D3.** Defina uma função `height` que retorna a altura da árvore sintáctica da sua entrada.

height : ArEx → NAT

Só isso mesmo.

(35) A

Defina os: `Nat`, `List`, `map`, `filter`, `fold`, `curry`, `uncurry`.
DEFINIÇÕES.

```
data Nat where
  O :: Nat
  S :: Nat → Nat

data List α where
  Empty :: List α
  Cons :: α → List α → List α

map :: (α → β) → List α → List β
map f (x : xs) = f : map f xs
map _ [] = []

curry :: (α × β → γ) → α → β → γ
curry f a b = f (a,b)

uncurry :: (α → β → γ) → α × β → γ
uncurry f (a,b) = f a b

filter :: (α → Bool) → List α → List α
filter _ [] = []
filter p (x : xs) = if p x then x : filter p xs
                    else filter p xs

fold :: (α → α → α) → α → List α → List α
fold b a (x : xs) = (a `b` x) `fold b a xs`
fold _ _ [] = []
```

estranha escolha de nomes

(32) B

Escolha **até 4** 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 (x : xs)(y : ys) = (x,y) : zip xs ys
zip _ _ = []

zipWith f (x : xs)(y : ys) = f x y : zipWith f xs ys

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

subseqs [] = [[]]
subseqs (x : xs) = map (x:) (subseqs xs) ++ subseqs xs

AUX:
[] ++ L = L
(x : xs) ++ L = x : (xs ++ L)
```

(12) C

Defina `zip` em termos da `zipWith` e `zipWith` em termos da `zip`.

```
zip = zipWith (λx y → (x,y))        zipWith f xs = map (λ(x,y) → f x y) . zip xs
```

(,)   uncurry f

D

Poderia usar (:+:), (:*:) para esses

Considere o tipo `Int` dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados `ArEx` para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias $(+)$ e $(\cdot)$ e a operação unária $(-_)$.

```
data ArEx where
  (+) :: Int → ArEx → ArEx
  (·) :: Int → ArEx → ArEx
  (-) :: Int → ArEx
  Int :: Int → ArEx
```

(2+3)·(7+1) ?

mas não para esse

(7) **D2.** Defina uma função `eval` de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

```
eval :: ArEx → Int
eval (n+e) = n+(eval e)
eval (n.e) = n·(eval e)
eval (-n)
  | n≥0 = 0-n
  | otherwise = n
```

(7) **D3.** Defina uma função `height` que retorna a altura da árvore sintáctica da sua entrada.

```
height :: ArEx → Nat
height (Int n) = 0
height (-n) = 0
height (x op e) = S(height e)
```

Só isso mesmo.

```
fold : (A → A → A) → A → List A → A
fold op i [] = i
fold op i (x :: xs) = op x (fold op i xs)
```

(35) **A**

Defina os: `Nat`, `List`, `map`, `filter`, `fold`, `curry`, `uncurry`.

DEFINIÇÕES.

```
data Nat where
  O : Nat
  S : Nat → Nat

List : Type → Type
data List A where
  [] : List A
  (::) : A → List A → List A

curry : ((A × B) → C) →
        (A → B → C)
curry f a b = f (a, b)

map : (A → B) → List A → List B
map _ [] = []
map f (x :: xs) = f x :: (map f xs)

filter : (A → Bool) → List A → List A
filter _ [] = []
filter p (x :: xs) =
  if p x
  then x :: (filter p xs)
  else filter p xs

uncurry : (A → B → C) → (A, B) → C
uncurry f (a, b) = f a b
```

(32) **B**

Escolha **até 4** 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.

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

countDown O = [O]
countDown (S n) = [S n] ++ (countDown n)

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

zipWith op xs = (map op) . zip xs
```

(12) **C**

Defina `zip` em termos da `zipWith` e `zipWith` em termos da `zip`.

```
zipWith op = (map op) . zip          zip = zipWith pair
```

```
pair : A → B → (A, B)
pair a b = (a, b)
```

(21) **D**

Considere o tipo `Int` dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados `ArEx` para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias $(+)$ e $(\cdot)$ e a operação unária $(-_)$.

```
Data ArEx where
    Uni : Int → ArEx
    Inv : Int → ArEx
    Add : Int → Int → ArEx
    Mul : Int → Int → ArEx
```

X (como aninhar?)

(7) **D2.** Defina uma função `eval` de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

```
eval : ArEx → Int
eval Uni z = z
eval (Inv z) = -(eval z)
eval (Add z z') = eval z + eval z'
eval (Mul z z') = eval z · eval z'
```

✓

(7) **D3.** Defina uma função `height` que retorna a altura da árvore sintáctica da sua entrada.

```
height : ArEx → Nat
height (Uni _) = 0
height (Inv z) = S0 + height z
height (Add z z') = S(max(height z, height z'))
height (Mul z z') = S(max(height z, height z'))
```

S ✓

```
max : (Nat, Nat) → Nat
max (0, m) = n
max (n, 0) = m
max (Sn, Sm) = S(max(n, m))
```

✓

Só isso mesmo.

misturou os dois formatos

(35) **A**

Defina os: `Nat`, `List`, `map`, `filter`, `fold`, `curry`, `uncurry`.

DEFINIÇÕES.

```
Data Nat where:
 O :: Nat
 SNat :: Nat → Nat

Data List where:
null :: List α
cons x null :: α → List α → List α
Cons :: α → List α → List α
Cons x null = [x]
Cons x xs = [x:xs]

filter :: (α → Boll) → List α → List α
filter p [x:xs] = | p x [x: filter p xs
                  | otherwise filter p xs

map :: (α → β) → List α → List β
map p [x:xs] = [p x : map p xs]

curry :: ((α × β) → ω) → α → β → ω
curry m n = curry (m,n)
uncurry :: (α → β → ω) → ((α×β) → ω)
uncurry (m,n) = uncurry m n
```

[] ? — nome muito ruim — ?? — ?? — ? — X

(32) **B**

Escolha **até 4** 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)]
zipW [] [] = []
zipW [x:xs] [] = [x: zipW xs]
zipW [] [x:xs] = [x: zipW xs]
zipW [x:xs] [y:ys] = [(x+y) : (zipW xs ys)]

Auxiliar
(+) Nat → Nat → Nat
n + O = n
n + Sm = S(n+m)

CountD O = [O]
CountD Sn = [Sn : Count n]

pairs [] = []
pairs [n] = [n]
pairs [n:(m:ms)] [(n,m) : pairs n:ms]
```

tu sempre cria singleton list [xs] em vez da lista xs!

type error

por quê?

(12) **C**

Defina `zip` em termos da `zipWith` e `zipWith` em termos da `zip`.

[x : xs] não é uma lista que começa com x, mas sim uma lista com único membro o x:xs.

(35) **A**

Defina os: `Nat`, `List`, `map`, `filter`, `fold`, `curry`, `uncurry`.

DEFINIÇÕES.

```
data Nat where
    0 : Nat
    S n : Nat → Nat
data List α where
    [] : List α
    (::) : α → List α → List α
map : (α→β) → List α → List β
map f [] = []
map f (x::xs) = f x :: map f xs

filter : (α → Bool) → List α → List α
filter p [] = []
filter p (x::xs)
    | p x = x :: filter p xs
    | otherwise = filter p xs
fold : α → (α→α→α) → List α → α
fold b o [] = b
fold b o (x::xs) = x `o` (fold b o xs)
curry : ((α , β) → γ) → (α → β → γ)
?
uncurry : (α → β → γ) → ((α, β) → γ)
?
```

nomes ruins!

(32) **B**

Escolha **até 4** 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 (x::xs) (y::ys) =
  if len (x::xs) == len (y::ys)
  then (x,y) :: zip xs ys
  else error "..."
zipW op [] [] = []
zipW op (x::xs) (y::ys) =
  if len (x::xs) == len (y::ys)
  then (x `op` y) :: zipW op xs ys
  else error "..."

countdown 0 = [0]
countdown Sn = Sn :: countdown n

any p [] = False
any p (x::xs) =
  if p x then True
  else any p xs

AUX:
len [] = 0
len (x::xs) = S (len x)
```

e se as listas são infinitas?

melhor parar do que explodir

booleanismo!

NUNCA!

(12) **C**

Defina `zip` em termos da `zipWith` e `zipWith` em termos da `zip`.

```
zip = zipW (,)
```

(21) **D**

Considere o tipo `Int` dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados `ArEx` para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias (+) e (·) e a operação unária (−_).

```
data ArEx where
    (+) : ArEx → ArEx → ArEx
    (·) : ArEx → ArEx → ArEx
    (−) : ArEx → ArEx
```

e o 42?

(7) **D2.** Defina uma função `eval` de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

```
eval : ArEx → (ArEx → ArEx → ArEx) → ArEx → Int
eval ar op ar = (eval (ar)) `op` (eval (ar))
eval . n = n

eval : (ArEx → ArEx) → ArEx
eval op ar = − ( eval ar)
```

X

(7) **D3.** Defina uma função `height` que retorna a altura da árvore sintáctica da sua entrada.

```
height : ArEx → Nat
```

Só isso mesmo.

assim é ≅Bool

(35) **A**

Defina os: `Nat`, `List`, `map`, `filter`, `fold`, `curry`, `uncurry`.

DEFINIÇÕES.

```
data Nat = O | S Nat

data List α = Nil | Cons ..?

map :: (α → β) → [α] → [β]
map _ [] = []
map f (x:xs) = f x : map f xs

filter :: (α → Bool) → [α] → [α]
filter _ [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs

fold :: α → (α → β → α) → [α] → [α]
fold r _ [] = r
fold r f (x:xs) = f x (fold r f xs)

curry :: (α × β → c) → (α → β → c)

uncurry :: (α → β → c) → (α × β → c)
```

(32) **B**

Escolha **até 4** 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 (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _ = []

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

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

countdown (S n) = S n : countdown n
countdown _ = [O]
```

(12) **C**

Defina `zip` em termos da `zipWith` e `zipWith` em termos da `zip`.

```
zipWith f xs ys = map f (zip xs ys)

zip xs ys = zipWith (,) xs ys

Aux: (,) :: α → β → (α,β)
(,) x y = (x,y)
```

uncurry f

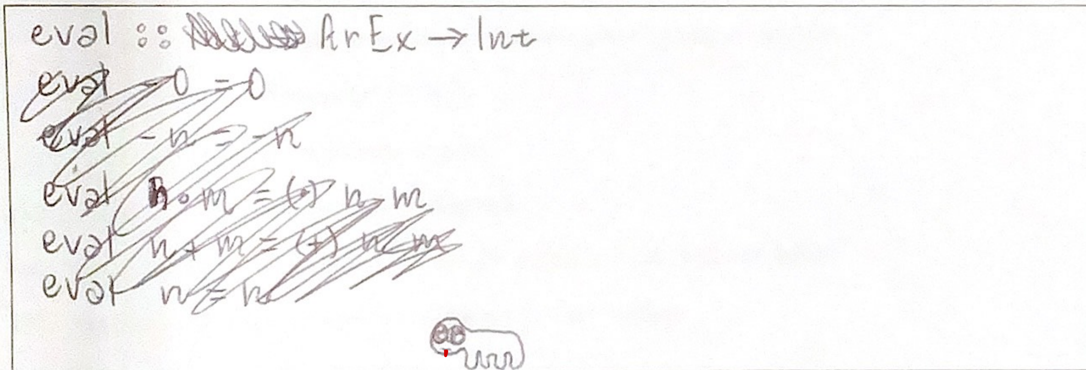
D

Considere o tipo `Int` dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados `ArEx` para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias (+) e (·) e a operação unária (−_).

ArEx :: Int → ArEx

(7) **D2.** Defina uma função `eval` de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.



(7) **D3.** Defina uma função `height` que retorna a altura da árvore sintáctica da sua entrada.

Só isso mesmo.

(35) **A**

Defina os: `Nat`, `List`, `map`, `filter`, `fold`, `curry`, `uncurry`.

DEFINIÇÕES.

```
Nat :: *
data Nat where
  O :: Nat
  S :: Nat -> Nat

List :: * -> *
data [α] where
  [] :: [α]
  (:) :: α -> [α] -> [α]

uncurry :: (α -> β -> γ) -> (α,β) -> γ
uncurry f (a,b) = f a b

map :: (α -> β) -> [α] -> [β]
map _ [] = []
map f (x:xs) = f x : map f xs

filter :: (α -> Bool) -> [α] -> [α]
filter _ [] = []
filter p (x:xs)
  | p x = x : filter p xs
  | otherwise = filter p xs

curry :: ((α,β) -> γ) -> α -> β -> γ
curry f a b = f (a,b)

fold :: (α -> α -> α) -> α -> [α] -> α
fold _ nEl [] = nEl
fold op nEl (x:xs) = x `op` (fold op nEl xs)
```

(32) **B**

Escolha **até 4** 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.

```
Countdown = ct
ct O = [O]
ct (S n) = (S n) : ct n

zip [] _ = []
zip _ [] = []
zip (a:as) (b:bs) = (a,b) : zip as bs

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

zipWith = zW
zW _ [] _ = []
zW _ _ [] = []
zW op (a:as) (b:bs) = (a `op` b) : zW op as bs

head :: [α] -> α   <- auxiliar
head [] = error "desc of nullary cons"
head (a:as) = a
```

(12) **C**

Defina `zip` em termos da `zipWith` e `zipWith` em termos da `zip`.

```
zip as bs = zipWith (\x -> \y -> (x,y)) as bs
zipWith op as bs = map (uncurry op) (zip as bs)
```

(21) **D**

Considere o tipo `Int` dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados `ArEx` para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias $(+)$ e $(\cdot)$ e a operação unária $(-_)$.

```
data ArEx where
  (:+) :: Int -> Int -> ArEx
  (:·) :: Int -> Int -> ArEx
  (:-) :: Int -> ArEx
  (:->) :: ArEx -> [ArEx] -> [ArEx]
```

e o (3+7)·2 já era...

???

(7) **D2.** Defina uma função `eval` de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

(7) **D3.** Defina uma função `height` que retorna a altura da árvore sintáctica da sua entrada.

Só isso mesmo.

(35) **A**

Defina os: Nat, List, map, filter, fold, curry, uncurry.

DEFINIÇÕES.

```
data Nat = O | S Nat

data List a = [] | (a : List a)

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

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

fold op b [] = b
fold op b (x : xs) = x `op` fold op b xs

curry :: ((a,b) -> c) -> a -> b -> c
curry f a b = f (a,b)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b) = f a b
```

(32) **B**

Escolha **até 4** 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 (x : xs) (y : ys) = (x,y) : zip xs ys
zip _ _ = []

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

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

countDown O = [O]
countDown (Sn) = (Sn) : countDown n
```

(12) **C**

Defina zip em termos da zipWith e zipWith em termos da zip.

```
AUXILIAR
pair :: a -> b -> (a,b)
pair a b = (a,b)

zip x y = zipWith pair x y
zipWith f x y = map (uncurry f) (zip x y)
```

(21) **D**

qual a definição?

Considere o tipo `Int` dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados `ArEx` para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias $(+)$ e $(\cdot)$ e a operação unária $(-_)$.

```
data ArEx = Int | -Int | Int :+ Int | Int :· Int
```

Neg

não seria o que tu imaginas isso

(7) **D2.** Defina uma função `eval` de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

```
eval :: ArEx → Int
```

(7) **D3.** Defina uma função `height` que retorna a altura da árvore sintáctica da sua entrada.

```
height :: ArEx → Nat
```

Só isso mesmo.

(35) **A**

Defina os: `Nat`, `List`, `map`, `filter`, `fold`, `curry`, `uncurry`.

DEFINIÇÕES.

```
inductive Nat : Type where
  | O : Nat
  | S : Nat → Nat

inductive List (α : Type) where
  | Nil : List α
  | Cons: α List α → List α
infix :60 "::" ⇒ Cons

map : (α → β) → List α → List β
  | _, .Nil ⇒ .Nil
  | f, x::xs ⇒ (f x)::(map f xs)

def filter : (α → Bool) → List α → List α
  |_, .Nil ⇒ .Nil
  | p, x::xs ⇒ if (p x) then x::(filter p xs)
                 else (filter p xs)

def fold : (α→α→α) → α → List α → α
  | op, e, x::xs ⇒ op x (fold op e xs)
  | _, e, _ ⇒ e

def curry : (α × β → γ) → α → β → γ
  | f, a, b ⇒ f ⟨a,b⟩

def uncurry : (α → β → γ) → α×β → γ
  | f, ⟨a,b⟩ ⇒ f a b
```

propaganda detectada

(32) **B**

Escolha **até 4** 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.

se fosse [1,2,3] ↦ [(1,1),(2,2),(3,3)] sim; mas é ↦ [(1,2),(2,3)]

```
① pairs = map Δ

② countdown O = [O]
   countdown (S n) = (S n)::(countdown n)

③ zip (x::xs) (y::ys) = (x,y)::(zip xs ys)
   zip _ _ = []

④ zipWith f (x::xs) (y::ys) = (f x y)::(zipWith f xs ys)
   zipWith _ _ _ = []

Auxiliar:
def Δ : α → α×α
  | a ⇒ ⟨a,a⟩
```

(12) **C**

Defina `zip` em termos da `zipWith` e `zipWith` em termos da `zip`.

```
zip = zipWith pac
def pac α → β → α×β
  | a, b ⇒ ⟨a,b⟩

zipWith f xs = map (uncurry f) ∘ zip xs
```

(21) **D**

Considere o tipo `Int` dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados `ArEx` para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias $(+)$ e $(\cdot)$ e a operação unária $(-_)$.

```
data ArEx : Type where
  Exadd : (Int → Int → Int) → Int → Int → ArEx
  Exneg : (Int → Int) → Int → ArEx
  Exmul : (Int × Int → Int) → Int → Int → ArEx
  Exatom : Int → ArEx
```

APENAS aqui!

(7) **D2.** Defina uma função `eval` de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

```
eval : ArEx → Int
eval (Exadd a b) = a+b
eval (Exneg a) = -a
eval (Exmul a b) = a*b
eval (Exatom a) = a
```

(7) **D3.** Defina uma função `height` que retorna a altura da árvore sintáctica da sua entrada.

```
height : ArEx → Nat
height (Exadd a b) = Succ ((height a) + (height b))
height (Exneg a) = Succ (height a)
height (Exmul a b) = Succ ((height a) + (height b))
height (Exatom a) = 0
```

max!

5 7 / 5+7 } tamanho 1

Só isso mesmo.

(35) **A**

Defina os: Nat, List, map, filter, fold, curry, uncurry.
DEFINIÇÕES.

```
Data Nat where
  O :: Nat
  S :: Nat -> Nat

Data List α where
  [ ] :: List α
  X:XS :: (α -> List α) -> List α

curry :: (a -> b -> c) -> (a -> b -> c))
curry f g h = f(g h)

uncurry :: (a -> (b -> c)) -> (a -> b -> c)
uncurry f (g h) = f g h

fold :: (α -> (α x α -> α)) -> (L α -> L α)

filter :: (α -> Bool) -> (L α -> L α)
filter p [] = []
filter p x:xs
  | p x = x : filter p xs
  | otherwise = filter p xs

map :: sorry
```

(32) **B**

Escolha **até 4** 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 :: List a -> List b -> List (a,b)
zip (x:xs) (y:ys) = XxY : zip xs ys
zip xs [] = xs
zip [] ys = ys

(*) :: Nat -> Nat -> Nat
_ * 0 = 0
m * (S m) = n + (n * m)

min :: Nat -> Nat -> Nat
min (S n) (S m) = min n m
min _ _ = 0

Odd :: Nat -> Bool
Odd O = False
odd (S n) = not (odd n)
```

(12) **C**

Defina zip em termos da zipWith e zipWith em termos da zip.

```
zip = zipwith (x) List a List b
zipwith = sorry
```

List α :: α -> (List α) -> α)   zip [1,2] [10,50] = [(1,10),(2,50)]   filter

cons   zipWith = [11,52]   (α -> Bool) -

Nil :: Nil [] = []   inits "hello" = ["", "h", "he" ..]   data Nat :: 0 | S

(35) **A**   Pairs [1,2,3,4] = [(1,2),(2,3)]

Defina os: Nat, List, map, filter, fold, curry, uncurry.
DEFINIÇÕES.

data Nat :: 0 | S ?

filter :: (α -> Bool) -> (List α) -> (List β) -

data List
Nil : []
Cons : α -> (List α) -> (List α)

X

tail
Tail [] = [X]
Tail [X s]

(32) **B**   ++

Escolha **até 4** 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.

(*)
_ * 0 = 0
0 * _ = 0
n * (S m) =
(n * m) + n

(+) :: Nat -> Nat -> Nat n + 0 = n ~ 0 + m = m ? n + (S m) = S (n + m)	min min 0 _ = 0 min _ 0 = 0 min (S n) (S m) = S (min m) × min!	
(*) _ * 0 = 0 ~ 0 * _ = 0 ? n * (S m) = (n * m) + n	Concat [] ys = ys type error X Concat (x :: xs) ys = x :: (xs ++ ys)	

(12) **C**

Defina zip em termos da zipWith e zipWith em termos da zip.

(35) **A**

Defina os: `Nat`, `List`, `map`, `filter`, `fold`, `curry`, `uncurry`.
DEFINIÇÕES.

```
Nat : Type
data Nat
  O : Nat
  S : Nat -> Nat

List : Type -> Type
data List α
  Nil : List α
  Cons : α -> List α -> List α

map : (α -> β) -> Lα -> Lβ
map _ [] = []
map f (x::xs) = (f x) :: (map f xs)

filter : (α -> Bool) -> Lα -> Lα   (abreviei)
filter _ [] = []
filter p (x::xs) = if (p x)
                     then x::xs'
                     else xs'
  where xs' = filter p xs

curry : (α × β -> γ) -> (α -> β -> γ)
curry f a b = f (a,b)

uncurry : (α -> β -> γ) -> (α × β -> γ)
uncurry f (a,b) = f a b

fold : (α -> α -> α) -> α -> Lα -> α
fold _ e [] = e
fold f e (x::xs) = f x (fold f e xs)
```

(32) **B**

Escolha **até 4** 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

countdown 0 = [0]
countdown (S n) = (S n) :: countdown n

zipWith f xs = map (uncurry f) . zip xs

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

(12) **C**

Defina `zip` em termos da `zipWith` e `zipWith` em termos da `zip`.

```
zip = zipWith (λx.λy.(x,y))     [anotação: (,)]
```

(21) **D**

Considere o tipo Int dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados **ArEx** para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias $(+)$ e $(\cdot)$ e a operação unária $(-_)$.

(7) **D2.** Defina uma função **eval** de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

(7) **D3.** Defina uma função **height** que retorna a altura da árvore sintáctica da sua entrada.

Só isso mesmo.

(35) **A**

Defina os: `Nat`, `List`, `map`, `filter`, `fold`, `curry`, `uncurry`.

DEFINIÇÕES.

```
data Nat = O | S Nat

data List a where
  Nil :: []
  Cons :: Cons a (Cons Nil)

map :: (a -> b) -> List a -> List b
map f (Cons x xs) = Cons (f x) (map xs)
map _ Nil = Nil

filter _ Nil = Nil
filter f (Cons x xs) = if f then (Cons x) (filter xs) else (filter xs)

curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

(32) **B**

Escolha **até 4** 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.

```
(+) n O = n
(+) n (S m) = S (n + m)

(*) _ O = O
(*) n SO = n
(*) n Sm = n + (n (*) m)

zip Nil Nil = Nil
zip (Cons x xs) Nil = Cons x xs
zip Nil (Cons y ys) = Cons y ys
zip (Cons x xs) (Cons y ys) = Cons (x,y) zip (xs, ys)

odd O = False
odd (S O) = True
odd n = ifThenElse (rem(n, SSO) == True) False True
```

(12) **C**

Defina `zip` em termos da `zipWith` e `zipWith` em termos da `zip`.

```
zipWith = add ((Cons (x,y)) zip(xs, ys))
zip = Cons (
```

(21) **D**

Considere o tipo `Int` dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados `ArEx` para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias $(+)$ e $(\cdot)$ e a operação unária $(-_)$.

```
data ArEx where:
(+) :: Int -> int -> int
(•) :: int -> int -> int
(−_) :: int -> int
```

X

(7) **D2.** Defina uma função `eval` de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

```
eval :: ArEx -> Int
```

X

(7) **D3.** Defina uma função `height` que retorna a altura da árvore sintáctica da sua entrada.

```
height :: ArEx -> Nat
```

X

Só isso mesmo.

(35) **A**

Defina os: Nat, List, map, filter, fold, curry, uncurry.
DEFINIÇÕES.

```
data Nat where
    O : Nat
    S : Nat -> Nat

data List a where
    [] : List a
    (::) : a -> List a -> List a

map : (α -> β) -> List α -> List β
map f [] = []
map f (x::xs) = f x :: map f xs

filter : (α -> Bool) -> List α -> List α
filter p [] = []
filter p (x::xs)
    | p x = x :: filter p xs
    | otherwise = filter p xs

fold : α -> (α -> α -> α) -> List α -> α
fold i op [] = i
fold i op (x::xs) = op x (fold i op xs)

curry : (α x β -> γ) -> α -> β -> γ
curry f a b = f (a,b)

uncurry : (α -> β -> γ) -> α x β -> γ
uncurry f (a,b) = f a b
```

(32) **B**

Escolha **até 4** 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

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

countdown O = [O]
countdown (S n) = S n :: countdown n

[] ++ xs = xs
(x::xs) ++ ys = x :: xs ++ ys   <- Aux

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

([x]++) ou ((++) [x])

ou, (BEM) melhor: (x:)

(faz sentido?)

(12) **C**

Defina zip em termos da zipWith e zipWith em termos da zip.

```
zip = zipWith (λx -> (λy -> (x,y)))
zipWith f xs = map (uncurry f) ∘ zip xs
```

1) D

Considere o tipo Int dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados ArEx para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias $(+)$ e $(\cdot)$ e a operação unária $(-_)$.

```
data ArEx where
  LiT : InT -> ArEx
  (:+) : ArEx -> ArEx -> ArEx
  (:•) : ArEx -> ArEx -> ArEx
  (:-) : ArEx -> ArEx
```

↳ isso seria infixo em Haskell (e binário).

(7) **D2.** Defina uma função eval de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

```
eval : ArEx -> InT
eval (LiT i) = i
eval (ex :+ ex') = eval ex + eval ex'
eval (ex :• ex') = eval ex • eval ex'
eval (:- ex) = - (eval ex)
```

(7) **D3.** Defina uma função height que retorna a altura da árvore sintáctica da sua entrada.

```
height : ArEx -> Nat
height (LiT i) = 0
height (ex :+ ex') = S (max (height ex) (height ex'))
height (ex :• ex') = S (max (height ex) (height ex'))
height (:- ex) = S (height ex)
```

Só isso mesmo.

(35) **A**

Defina os: `Nat`, `List`, `map`, `filter`, `fold`, `curry`, `uncurry`.

DEFINIÇÕES.

```
Nat :: *
data Nat where
  O :: Nat
  S :: Nat → Nat

List :: * → *
data List α where
  Nil :: List α
  cons :: α → List α → List α

fold :: α → (α → α → α) → List α → List α
fold r b [] = r
fold r b (x:xs) = b x (fold r b xs)

map :: (α → β) → List α → List α
map f [] = []
map f (x:xs) = f x : map f xs

filter :: (α → Bool) → List α → List α
filter p [] = []
filter p (x:xs)
  | p x = p : filter p xs
  | otherwise = filter p xs

curry :: (α × β → γ) → (α → β → γ)
curry f = λx λy . f(x,y)
uncurry :: (α → β → γ) → α × β → γ
uncurry f = λ(x,y) . f x y
```

(32) **B**

Escolha **até 4** 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 (a:as)(b:bs) = (a,b) : (zip as bs)

zipWith f xs = map (uncurry f) . zip xs

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

countDown :: Nat → List Nat
cD O = [O]
cD (Sn) = Sn : (cD n)
```

isso!

(12) **C**

Defina `zip` em termos da `zipWith` e `zipWith` em termos da `zip`.

```
zipWith f xs = map (uncurry f) . zip xs
zip = zipWith (λa λb . (a,b))
```

```
f : α → β → γ
uncurry f : α × β → γ
map uncurry f : List α × β → List γ
```

(21) **D**

esses seriam as folhas na vdd.

Considere o tipo `Int` dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados `ArEx` para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias $(+)$ e $(\cdot)$ e a operação unária $(-_)$.

```
data ArEx were
    Node :: Int → ArEx
    Un :: (Int → Int) → ArEx → ArEx
    bin :: (Int → Int → Int) → ArEx → ArEx → ArEx
```

(7) **D2.** Defina uma função `eval` de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

```
Eval :: ArEx → Int
Eval (Node x) = x
Eval (Un op x) = op (Eval x)
Eval (bin op x y) = op (Eval x) (Eval y)
```

(7) **D3.** Defina uma função `height` que retorna a altura da árvore sintáctica da sua entrada.

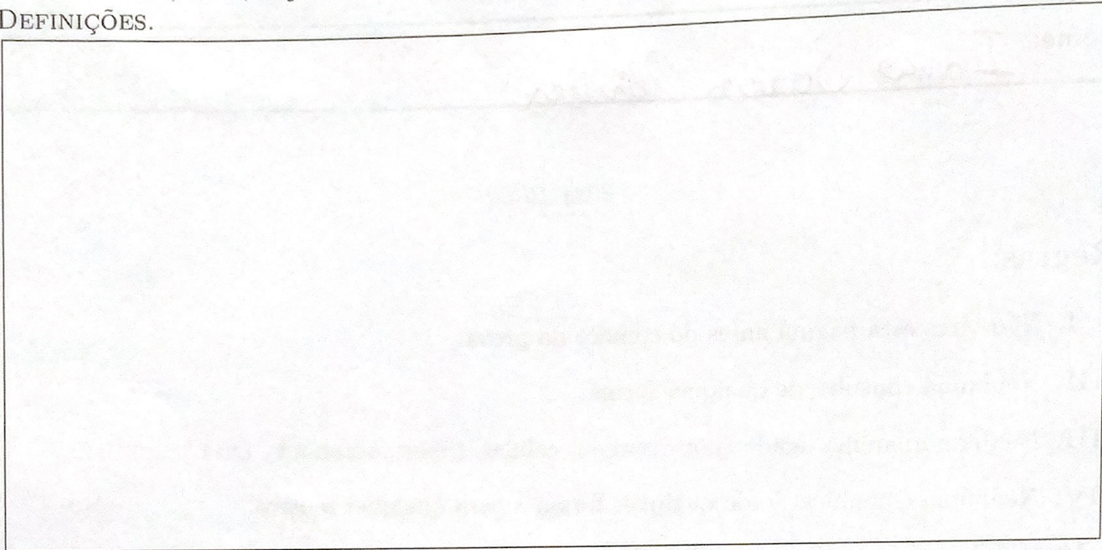
```
height :: ArEx → Nat
height (Node _) = 0
height (Un _ x) = S (height x)
height (bin _ x y) = S (max (height x, height y))

Max :: Nat x Nat → Nat
Max n 0 = n
Max 0 n = n
Max (Sn)(Sm) = S(max(n,m))
```

✓

Só isso mesmo.

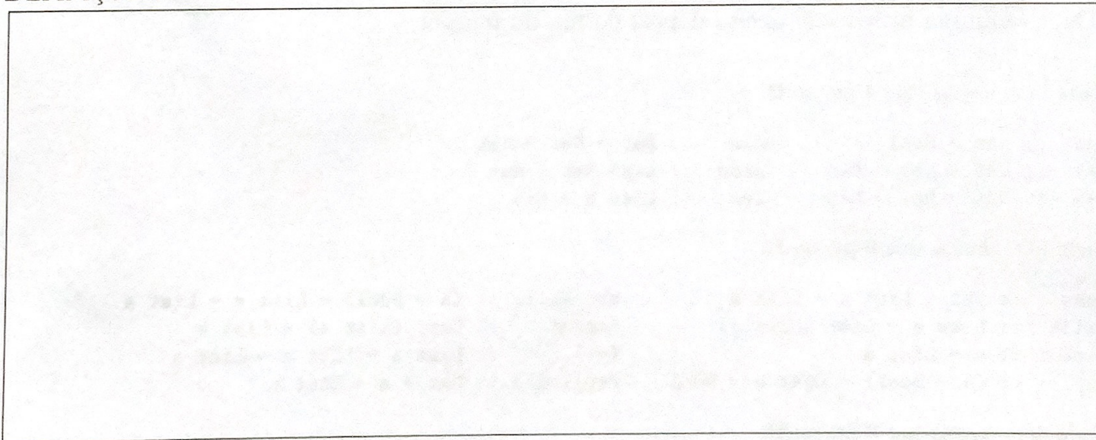
(35) **A**

Defina os: `Nat`, `List`, `map`, `filter`, `fold`, `curry`, `uncurry`.
DEFINIÇÕES.



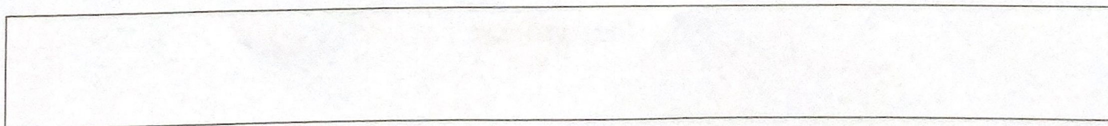
(32) **B**

Escolha **até 4** 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) **C**

Defina `zip` em termos da `zipWith` e `zipWith` em termos da `zip`.



(35) A

Defina os:

`Nat`,
`List`,
`map`,
`filter`,
`fold`,
`curry`,
`uncurry`.

(32) B

Escolha **até 4** 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.

A

curry :: ((α × β) -> γ) -> (α -> β -> γ)
uncurry :: (α -> β -> γ) -> (α × β) -> γ

fold :: α × (α × α -> α) × List α -> α

Nat :: type

Nat = zero | Succ Nat

List :: type -> type

List = [ ]α | List α ?

filter :: (α -> Bool) × List α -> List α

map :: (α -> β) × List α -> List β

fold r b r = [ ]
fold r b [x:xs] = x b fold r b [xs]

necessário

filter p [ ] = [ ]
filter p [x:xs]
| p x = True, cons x filter p [xs]
| otherwise, filter p [xs]

map p [ ]α = [ ]β
map p [x:xs] = p x :: map p [xs]

mesma confusão sobre [x:xs] e [xs].

nome ruim

B

```
zip [ ] [ ] = [ ]

zip List[a:as] [ ] = List[a:as]

zip [ ] List[b:bs] = List[b:bs]

zip List[a:as] List[b:bs] = (a,b) : zipList[as] List[bs]
```

```
pairs [ ] = [ ]
pairs [x] = [ ]
pairs [x:xs] = (x : head[xs]) : pairs[xs:xss]
```

```
tails [ ]α = [ ][ ]α
tails [x:xs] = [x::[ ]] :: tails [xs]
```

```
concat [ ][ ]α = [ ]α
concat [x:xs] = head[x] : concat [xs]
```

aux

```
head [ ] = error
head [x:xs] = x
```

Correções em vermelho: ?, ?, ?, X, X, ✓, quem é?, ",", sinal ruim!, [ ] (por quê?), xs, X

(12) **C**

Defina `zip` em termos da `zipWith` e `zipWith` em termos da `zip`.

C

zipwith (+) _ [ ] = [ ]

zipwith (+) [x:xs] = head[zip[v:xs]] + head[zip[xs]] : zipwith(+) [x s::xss]

(21) **D**

Considere o tipo `Int` dado, junto com suas operações.

(7) **D1.** Defina um tipo de dados `ArEx` para representar expressões de aritmética de inteiros formadas apenas pelas operações binárias (+) e (·) e a operação unária (−_).

(7) **D2.** Defina uma função `eval` de *evaluate* que dada uma expressão aritmética de inteiros retorna seu valor-resultado.

(7) **D3.** Defina uma função `height` que retorna a altura da árvore sintáctica da sua entrada.

D1

ArEx :: (int -> int -> int) -> (int -> int) -> int

ArEx ab = λa. λb (5a8 + b7) ??

D2

D2 eval :: ArEx

eval a,b = 5a8 + b7