2010-1

```
-- 2

data Tree t = NilT | Node t (Tree t) (Tree t) deriving (Show)

dfs :: (Show t) => (Tree t) -> IO()
dfs NilT = do return ()
dfs (Node a b c) = do putStr (show a)
                      printTree [b, c]

printTree :: (Show t) => [(Tree t)] -> IO()
printTree [] = do putStr ".\n"
printTree ((NilT):as) = printTree as
printTree ((Node a b c):as) = do putStr ", "
                                 putStr (show a)
                                 printTree ([b, c]++as)

ex = Node 10 (Node 7 NilT NilT) (Node 4 (Node 12 NilT (Node 14 NilT NilT)) (Node 6 (Node 2
NilT NilT) (Node 8 NilT NilT)))

-- 3

-- (a) map.map.foldr
{-
(.) :: (y -> z) -> (x -> y) -> (x -> z)
map :: (a -> b) -> [a] -> [b]
foldr :: (c -> d -> d) -> d -> [c] -> d

-- (i) map.map

(y -> z) => (a -> b) -> [a] -> [b]
y => (a -> b)
z => [a] -> [b]

(x -> y) => (t -> v) -> [t] -> [v]
x => (t -> v)
y -> [t] -> [v]

a => [t]
b => [v]
```

```
z => [[t]] -> [[v]]
(x -> z) => (t -> v) -> [[t]] -> [[v]]

-- (ii) map.map.foldr

(.) :: (y -> z) -> (x -> y) -> (x -> z)

(y -> z) => (t -> v) -> [[t]] -> [[v]]
y => (t -> v)
z => [[t]] -> [[v]]

(x -> y) => (c -> d -> d) -> d -> [c] -> d
x => (c -> d -> d)
y -> d -> [c] -> d

t => d
v => [c] -> d
z => [[d]] -> [[[c] -> d]]

(x -> z) => (c -> d -> d) -> [[d]] -> [[[c] -> d]]
-}

-- (b) map.((.) (foldr (++) (foldr (++) [] [[1], [2], [4,5,6], [3]])))
{-
(foldr (++) [] [[1], [2], [4,5,6], [3]]) :: [Int]

map.((.) (foldr (++) (foldr (++) [] [[1], [2], [4,5,6], [3]]))) => map.((.) (foldr (++) [Int]))

map.((.) (foldr (++) [Int]))

(.) :: (y -> z) -> (x -> y) -> (x -> z)
foldr :: (t -> u -> u) -> u -> [t] -> u
map :: (a -> b) -> [a] -> [b]
(++) :: [v] -> [v] -> [v]

-- foldr (++) [Int]

(t -> u -> u) => [v] -> [v] -> [v]
t => [v]
u => [v]

foldr (++) :: [v] -> [[v]] -> [v]
foldr (++) [Int] :: [[Int]] -> [Int]
```

```
-- (.) foldr (++) [Int]
(.) (y -> z) -> (x -> y) -> (x -> z)

(y -> z) => [[Int]] -> [Int]
y => [[Int]]
z => [Int]

(.) foldr (++) [Int] => (p -> [[Int]]) -> (p -> [Int])

-- map.((.) foldr (++) [Int])

(y -> z) => (a -> b) -> [a] -> [b]
y => (a -> b)
z => [a] -> [b]

(x -> y) => (p -> [[Int]]) -> (p -> [Int])
x => (p -> [[Int]])
y => (p -> [Int])

a => p
b => [Int]
z => [p] -> [[Int]]

(x -> z) => (p -> [[Int]]) -> [p] -> [[Int]]

map.((.) foldr (++) [Int]) :: (p -> [[Int]]) -> [p] -> [[Int]]
-}

-- 4

data No t = No t [t]
data Grafo t = Grafo [No t]

getNeighborhood :: (Eq t) => [(No t)] -> t -> [t]
getNeighborhood [] _ = []
getNeighborhood ((No v adj):graph) no = if v == no
                                        then adj
                                        else getNeighborhood graph no

mapEdges :: (Int -> Int -> Int) -> (Grafo Int) -> (Int -> [Int])
mapEdges f (Grafo graph) no = [f no x | x <- adj]
   where adj = getNeighborhood graph no
```

```
gr :: Grafo Int
gr = Grafo [(No 10 [7, 4]), (No 4 [12, 6]), (No 12 [14]), (No 6 [2, 8])]
```

2010-2

```
-- 2

--(a)

data Habitantes = Elfo String | Humano String | Anao String | Hobbit String deriving (Eq)

--(b)

class Comp t where
        maisImportante :: t -> t -> Bool

--(c)

instance Comp Habitantes where
        maisImportante (Elfo n1) (Elfo n2) = n1 > n2
        maisImportante (Humano n1) (Humano n2) = n1 > n2
        maisImportante (Anao n1) (Anao n2) = n1 > n2
        maisImportante (Hobbit n1) (Hobbit n2) =  n1 > n2
        maisImportante (Elfo _) _ = True
        maisImportante (Humano _) (Elfo _) = False
        maisImportante (Humano _) _ = True
        maisImportante (Anao _) (Elfo _) = False
        maisImportante (Anao _) (Humano _) = False
        maisImportante (Anao _) _ =True
        maisImportante (Hobbit) _ = False

-- 3

--(a)

data Arvore t = No t (Maybe (Arvore t)) (Maybe (Arvore t)) deriving (Show)

--(b)

criarArvoreDeImportancia :: (Comp t) => [t] -> Maybe (Tree t)
criarArvoreDeImportancia [] = Nothing
```

```
criarArvoreDeImportancia (a:as) = insertList (Just (No a Nothing Nothing)) as
   where
         insertList tree [] = tree
         insertList (Just (No a b c)) (v:vs) = insertList (insertVal v (Just (No a b c))) vs
         insertVal Nothing v = (Just Node (v Nothing Nothing))
       insertVal (Just (No a b c)) v = if (maisImportante v a == True)
                                  then (Just (No a b (insertVal c v)))
                                  else (Just (No a (insertVal b v) c))

--(c)

filhos :: Maybe (Tree Habitantes) -> Habitantes -> [Habitantes]
filhos Nothing _ = []
filhos (Just (Node a b c)) hbt | a == hbt = toList c
                           | maisImportante hbt a == True = filhos c hbt
                           | otherwise = filhos b hbt
   where
         toList Nothing = []
         toList (Just (Node a b c)) = [a]++(toList b)++(toList c)

-- 4

--(a) (+ 2).((*) 2).(2 /)

(+ 2).((*) 2).(2 /)

(.) : (y -> z) -> (x -> y) -> (x -> z)
(+2) :: (Num a) => a -> a
(*2) :: (Num b) => b -> b
(2/) :: (Fractional c) => c -> c -> c

-- (i) (+2).((*) 2)

(y -> z) => a -> a
y => a
z => a

(x -> y) => b -> b -> b
x => b
y => b

a => b
```

(x -> z) => b -> b

(+2).((*) 2) :: (Num b) => b -> b

-- (ii) (+2).((*) 2).(2 /)

(y -> z) => b -> b
y => b
z => b

(x -> y) => c -> c
x => c
y => c

(x -> z) => c -> c

(+2).((*) 2).(2 /) :: (Fractional c) => c -> c

-- (b) (.).filter

(.) :: (t -> v) -> (s -> t) -> (s -> v)
filter :: (a -> Bool) -> [a] -> [a]

(y -> z) => (t -> v) -> (s -> t) -> (s -> v)
y => (t -> v)
z => (s -> t) -> (s -> v)

(x -> y) => (a -> Bool) -> [a] -> [a]
x => (a -> Bool)
y => [a] -> [a]

t => [a]
v => [a]

z => (s -> [a]) -> (s -> [a])
(x -> z) => (a -> Bool) -> (s -> [a]) -> (s -> [a])

(.).filter :: (a -> Bool) -> (s -> [a]) -> (s -> [a])

2011-1

--1

```
-- (a)

data Tree t = Nilt | Node t (Tree t) (Tree t) deriving (Show, Eq)

-- (b)

insertValue :: (Ord t) => t -> (Tree t) -> (Tree t)
insertValue v Nilt = (Node v Nilt Nilt)
insertValue v (Node a b c) = if v >= a
                        then (Node a b (insertValue v c))
                                                        else (Node a (insertValue v b) c)

buildSearchTree :: (Ord t) => [t] -> Tree t
buildSearchTree [] = Nilt
buildSearchTree (a:as) = insertList as (Node a Nilt Nilt)
   where insertList [] tree = tree
         insertList (x:xs) tree = insertList xs (insertValue x tree)

-- (c)

inorder :: (Tree t) -> [t]
inorder Nilt = []
inorder (Node a b c) = (inorder b) ++ [a] ++ (inorder c)

searchTreeSort :: (Ord t) => [t] -> [t]
searchTreeSort xs = inorder (buildSearchTree xs)

-- 2

-- (a)

f::Ord a => [a] -> (a -> [b]) -> [[b]]

f.map (+1)

(+1) :: (Num t) => t -> t
map :: (u -> v) -> [u] -> [v]
(.) :: (y -> z) -> (x -> y) -> (x -> z)

map (+1)

(u -> v) => (t -> t)
```

```
u => t
v => t

map (+1) => (Num t) [t] -> [t]

(y -> z) => [a] -> (a -> [b]) -> [[b]]
y => [a]
z => (a -> [b]) -> [[b]]

(x -> y) => ([t] -> [t])
x => [t]
y => [t]

t => a

z => (t -> [b]) -> [[b]]

(x -> z) => [t] -> (t -> [b]) -> [[b]]

f.map (+1) :: [t] -> (t -> [b]) -> [[b]]

-- (b) ((:).foldr.foldr) (:)

(.) :: (y -> z) -> (x -> y) -> (x -> z)
(:) :: a -> [a] -> [a]
foldr :: (t -> u -> u) -> u -> [t] -> u

-- (i) (:).foldr

(y -> z) => a -> [a] -> [a]
y => a
z => [a] -> [a]

(x -> y) => (t -> u -> u) -> u -> [t] -> u
x => (t -> u -> u)
y => u -> [t] -> u

a => u -> [u] -> u
z => [u -> [t] -> u] -> [u -> [t] -> u]

(x -> z) => (t -> u -> u) -> ([u -> [t] -> u] -> [u -> [t] -> u])

(:).foldr :: (t -> u -> u) -> ([u -> [t] -> u] -> [u -> [t] -> u])
```

```
-- (ii) (:).foldr.foldr

(y -> z) => (t -> u -> u) -> ([u -> [t] -> u] -> [u -> [t] -> u])
y => (t -> u -> u)
z => [u -> [t] -> u] -> [u -> [t] -> u]

(x -> y) => (m -> n -> n) -> n -> [m] -> n
x => (m -> n -> n)
y => n -> [m] -> n

t => n
u => [m]
u => n
n => [m]

x => (m -> [m] -> [m])
z => [[m] -> [[m]] -> [m]] -> [[m] -> [[m]] -> [m]]

x -> z => (m -> [m] -> [m]) -> [[m] -> [[m]] -> [m]] -> [[m] -> [[m]] -> [m]]
(:).foldr.foldr => (m -> [m] -> [m]) -> [[m] -> [[m]] -> [m]] -> [[m] -> [[m]] -> [m]]

-- (iii) ((:).foldr.foldr) (:)

(:) :: a -> [a] -> [a]

m => a
[m] => [a]
[m] => [a]

((:).foldr.foldr) (:) => (a -> [a] -> [a]) -> [[a] -> [[a]] -> [a]] -> [[a] -> [[a]] -> [a]]

-- 3
```

2012 - 1

```
-- 1
data T t = F | R t (T t) (T t) deriving (Show, Eq)

(***) ::  String -> String -> String
(***) a b = a ++ " " ++ b
```

```
g :: (t -> String) -> (T t) -> (T String)
g _ F = F
g i ( R a e d) = R ( i a ) ( g i e) ( g i d)

h :: (String -> String -> String) -> (T String) -> String -> String
h j F l = l
h j (R a e d) l | ( R a e d) == ( R " " F F ) = "" ++ (h j e l) `j` (h j d l)
                | otherwise = a `j` ( h j e l) `j` ( h j d l)

f :: (T t) -> ((t -> String) -> (T t) -> (T String)) -> ((String -> String -> String) -> (T String) ->
String -> String)
f F _ _ = []
f x m n = n (***) ( m show x ) []

result :: String
result = f ( R 1 ( R 2 F F ) ( R 3 F F)) g h


(.) :: (y->z) -> (x->y) -> (x->z)
map :: (a->b) -> [a] -> [b]
f :: (T t) -> ((t -> String) -> (T t) -> (T String)) -> ((String -> String -> String) -> (T String) ->
String -> String)
F : T t

map.f ::

(y->z) => (a->b) -> [a] -> [b]
(x->y) => (T t) -> ((t -> String) -> (T t) -> (T String)) -> ((String -> String -> String) -> (T String)
-> String -> String)
y => (a->b)
z => [a] -> [b]

x => (T t)
y => ((t -> String) -> (T t) -> (T String)) -> ((String -> String -> String) -> (T String) -> String ->
String)

a => ((t -> String) -> (T t) -> (T String))
b => ((String -> String -> String) -> (T String) -> String -> String)

z => [((t -> String) -> (T t) -> (T String))] -> [((String -> String -> String) -> (T String) -> String
-> String)]
```

```
map.f => (T t) -> [((t -> String) -> (T t) -> (T String))] -> [((String -> String -> String) -> (T String) -> String -> String)]

(map.f) F => [((t -> String) -> (T t) -> (T String))] -> [((String -> String -> String) -> (T String) -> String -> String)]

-- 2

data No t = No t [t] deriving (Show)
data Grafo t = Grafo [No t] deriving (Show)

popularGrafo :: (Eq t) => [t] -> [(t, t)] -> Grafo t
popularGrafo rot arestas = Grafo grafo
   where grafo = [No x [b | (a, b) <- arestas, a == x] | x <- rot]

getAdjacentes :: (Eq t) => t -> Grafo t -> [t]
getAdjacentes no (Grafo []) = []
getAdjacentes no (Grafo ((No a adj):xs)) = if no == a
                                    then adj
                                                                                    else getAdjacentes
no (Grafo xs)

buscaEmLargura :: (Eq t) => Grafo t -> t -> t -> Bool
buscaEmLargura grafo ini fim = aux [ini] grafo fim
   where aux [] _ _ = False
         aux (x:xs) grafo fim = if x == fim
                                     then True
                                                                          else aux (xs++(getAdjacentes x
grafo)) grafo fim
```

2013-1

```
type Nome = String
type Ano = Int
type Artista = String
type Publisher = String

type Elem t = (Int, t)

data Dica =  Cons Dica Dica | Nilt | Filmes Nome Ano | Musicas Nome Ano Artista | Jogo Nome Ano Publisher deriving (Show, Eq)
```

```
data Fila (Elem t) = Nilt | Cons (Elem t) (Fila (Elem t))


-- começa com o
inserir :: Fila (Elem t) -> t -> Int -> Fila (Elem t)
inserir Nilt val idf = Cons (idf, val) Nilt
inserir (Cons he ta) val idf  = Cons he (inserir ta val (idf+1))

remover :: Fila t -> Fila t
remover Nilt = Nilt
remover (Cons a b) = b

pesquisar :: Fila t -> Int -> Bool
pesquisar Nilt _ = False
pesquisar (Cons (Elem (a,b)) c) idf
                                    | a == idf = True
                                    | otherwise = pesquisar c


consultar :: Dica -> (Dica -> Dica -> Int -> Bool) -> Int -> [Dica]
consultar dica simi limi (Cons (Elem (idf, val)) final) =
consultar dica simi limi


---------------------------------------------------------- INFERÊNCIA-----------------------------------------------

--a)

((:).foldr.foldr) (:)

Dados {
        (:) :: a -> [a] -> [a]
        (.) :: (b -> c) -> (d -> b) -> d -> c
        foldr :: (e -> f -> f) -> f -> [e] -> f
}

(b->c) = a -> [a] -> [a]
b = a
c = [a] -> [a]

(d -> b) = (e -> f -> f) -> f -> [e] -> f
d = (e -> f -> f)
b = f -> [e] -> f
```

```
a = f -> [e] -> f

(:).foldr :: d -> c
(:).foldr :: (e -> f -> f) -> [a] -> [a]
(:).foldr :: (e -> f -> f) -> [f -> [e] -> f] -> [f -> [e] -> f]

(b -> c) = (e -> f -> f) -> [f -> [e] -> f] -> [f -> [e] -> f]
b = (e -> f -> f)
c = [f -> [e] -> f] -> [f -> [e] -> f]

(d -> b) = (x -> y -> y) -> y -> [x] -> y
d = (x -> y -> y)
b = y -> [x] -> y

(e -> f -> f) = y -> [x] -> y
e = y
f = [x]
f = y
y = [x]

(:).foldr.foldr :: d -> c
(:).foldr.foldr ::(x -> y -> y) -> [f -> [e] -> f] -> [f -> [e] -> f]
(:).foldr.foldr ::(x -> [x] -> [x]) -> [[x] -> [[x]] -> [x]] -> [[x] -> [[x]] -> [x]]

(x -> [x] -> [x]) = a -> [a] -> [a]
x = a
[x] = [a]
[x] = [a]

((:).foldr.foldr) (:) :: [[a] -> [[a]] -> [a]] -> [[a] -> [[a]] -> [a]]

--b)

map.map.foldr

Dados {
        map :: (a -> b) -> [a] -> [b]
        (.) :: (y -> z) -> (x -> y) -> x -> z
        foldr :: (e -> f -> f) -> f -> [e] -> f
}

(y -> z) = (a -> b) -> [a] -> [b]
```

```
y = (a -> b)
z = [a] -> [b]

(x -> y) = (e -> f -> f) -> f -> [e] -> f
x = (e -> f -> f)
y = f -> [e] -> f

(a -> b) = f -> [e] -> f
a = f
b = [e] -> f

map.foldr :: x -> z
map.foldr :: (e -> f -> f) -> [f] -> [b]
map.foldr :: (e -> f -> f) -> [f] -> [[e] -> f]

(y -> z) = (a -> b) -> [a] -> [b]
y = (a -> b)
z = [a] -> [b]

(x -> y) = (e -> f -> f) -> [f] -> [[e] -> f]
x = (e -> f -> f)
y = [f] -> [[e] -> f]

(a -> b) = [f] -> [[e] -> f]
a = [f]
b = [[e] -> f]

map.map.foldr :: x -> z
map.map.foldr :: (e -> f -> f) -> [[f]] -> [[[e] -> f]]

--c)

foldr.foldr.foldl

Dados {
        foldl :: (a -> b -> a) -> a -> [b] -> a
        (.) :: (y -> z) -> (x -> y) -> x -> z
        foldr :: (e -> f -> f) -> f -> [e] -> f
}

(y -> z) = (e -> f -> f) -> f -> [e] -> f
y = (e -> f -> f)
z = f -> [e] -> f
```

(x -> y) = (a -> b -> a) -> a -> [b] -> a
x = (a -> b -> a)
y = a -> [b] -> a

(e -> f -> f) = a -> [b] -> a
e = a
f = [b]
f = a
a = [b]

foldr.foldl :: x -> z
foldr.foldl :: (a -> b -> a) -> f -> [e] -> f
foldr.foldl :: ([b] -> b -> [b]) -> [b] -> [[b]] -> [b]

(y -> z) = (e -> f -> f) -> f -> [e] -> f
y = (e -> f -> f)
z = f -> [e] -> f

(x -> y) = ([b] -> b -> [b]) -> [b] -> [[b]] -> [b]
x = ([b] -> b -> [b])
y = [b] -> [[b]] -> [b]

(e -> f -> f) = [b] -> [[b]] -> [b]
e = [b]
f = [[b]]
f = [b]
[[b]] = [b]
Logo, deu pau! Pois não tem como [b] ser igual a [[b]].

--d)

foldl.foldr.foldl

Dados {
	foldl :: (a -> b -> a) -> a -> [b] -> a
	(.) :: (y -> z) -> (x -> y) -> x -> z
	foldr :: (e -> f -> f) -> f -> [e] -> f
}

(y -> z) = (a -> b -> a) -> a -> [b] -> a
y = (a -> b -> a)

```
z = a -> [b] -> a

(x -> y) = (e -> f -> f) -> f -> [e] -> f
x = (e -> f -> f)
y = f -> [e] -> f

(a -> b -> a) = f -> [e] -> f
a = f
b = [e]
a = f

foldl.foldr :: x -> z
foldl.foldr :: (e -> f -> f) -> f -> [[e]] -> f

(y -> z) = (e -> f -> f) -> f -> [[e]] -> f
y = (e -> f -> f)
z = f -> [[e]] -> f

(x -> y) = (a -> b -> a) -> a -> [b] -> a
x = (a -> b -> a)
y = a -> [b] -> a

(e -> f -> f) = a -> [b] -> a
e = a
f = [b]
f = a
a = [b]

foldl.foldr.foldl :: x -> z
foldl.foldr.foldl :: ([b] -> b -> [b]) -> [b] -> [[[b]]] -> [b]

--e)

map.(foldr(/2))

Dados {
        2 :: Num a => a
        map :: (a -> b) -> [a] -> [b]
        (/) :: Fractional w => w -> w -> w
        (.) :: (y -> z) -> (x -> y) -> x -> z
        foldr :: (e -> f -> f) -> f -> [e] -> f
}
```

```
(/2) :: (Fractional w) => w -> w

(e -> f -> f) = w -> w
e = w
(f -> f) = w

foldr(/2) :: (Fractional w) => f -> [e] -> f
foldr(/2) :: (Fractional (f -> f)) => f -> [(f -> f)] -> f

(y -> z) = (a -> b) -> [a] -> [b]
y = (a -> b)
z = [a] -> [b]

(x -> y) = f -> [(f -> f)] -> f
x = f
y = [(f -> f)] -> f

(a -> b) = [(f -> f)] -> f
a = [f -> f]
b = f

map.(foldr(/2)) :: x -> z
map.(foldr(/2)) :: f -> [a] -> [b]
map.(foldr(/2)) ::(Fractional (f -> f)) => f -> [[f -> f]] -> [f]
```