JSON:

```
data Value = Vint Int | Vstring String | Vbool Bool | Vlist [Value]
type Field = String
data Svson i = Empty | Obj i (Field -> Maybe Value) (Svson i) (Svson i)

-- La estructura Svson i representa un árbol binario de búsqueda (BST)
donde cada nodo contiene:
-- Un índice de tipo i (que debe ser ordenable)
-- Una función que mapea nombres de campos a valores opcionales
-- Dos subárboles izquierdo y derecho

-- Funciones auxiliares implementadas
valuesOf :: [Field] -> (Field -> Maybe Value) -> [Value]
-- Obtiene los valores de una lista de campos de un objeto

valuesWithFields :: [Field] -> (Field -> Maybe Value) -> [(Field,
Value)]
-- Obtiene pares (campo, valor) para los campos que existen

only :: [Field] -> (Field -> Maybe Value) -> (Field -> Maybe Value)
-- Restringe un objeto a solo los campos especificados

update :: Field -> Value -> (Field -> Maybe Value) -> (Field -> Maybe
Value)
-- Actualiza o agrega un campo con un valor en un objeto

singleton :: Field -> Value -> (Field -> Maybe Value)
-- Crea un objeto con un solo campo y valor

-- Implementar
-- devuelve una lista con todos los índices presentes en la estructura,
en cualquier orden
indices :: Svson i -> [i]
indices Empty = []
indices (Obj i f sv1 sv1) = i : (indices sv1 ++ indices sv2)

-- indica si un índice dado existe en la estructura. Debe aprovechar la
propiedad de BST para ser eficiente.
belongs :: Ord i => i -> Svson i -> Bool
belongs x Empty = False
belongs x (Obj i f sv1 sv2) = if i == x then True
                              else if i < x then belongs x sv1
                                            else belongs x sv2

-- busca un índice específico y devuelve los valores de los campos
solicitados para ese índice. Si el índice no existe, devuelve una lista
vacía.
lookupProjecting :: Ord i => i -> [Field] -> Svson i -> [Value]
lookupProjecting x fs Empty = []
lookupProjecting x fs (Obj i f sv1 sv2) = if x == i then valuesOf fs f
                                                    else if x < i
```

```
                                                                then
lookupProjecting x fs sv1
                                                                else
lookupProjecting x fs sv2

-- actualiza o inserta un campo en el objeto con el índice dado. Si el
índice no existe, crea un nuevo nodo en la posición correcta del BST.
upsert :: Ord i => i -> Field -> Value -> Svson i -> Svson i
upsert x f v Empty = Obj x (singleton f v) Empty Empty
upsert x f v (Obj i sv1 sv2) = if x == i then Obj (update f v i) sv1
sv2
                                  else if
```

SVSON:

```
data Value = Vint Int | Vstring String | Vbool Bool | Vlist [Value]
    deriving Show
type Field = String
data Svson i = Empty | Obj i (Field -> Maybe Value) (Svson i) (Svson i)


-- La estructura Svson i representa un árbol binario de búsqueda (BST)
donde cada nodo contiene:
-- Un índice de tipo i (que debe ser ordenable),
-- Una función que mapea nombres de campos a valores opcionales,
-- Dos subárboles izquierdo y derecho

-- Obtiene los valores de una lista de fields de un objeto
valuesOf :: [Field] -> (Field -> Maybe Value) -> [Value]
valuesOf = foldr (\k f -> \g -> case g k of
                                    Nothing -> f g
                                    Just v  -> v : f g)
                (const [])

-- Obtiene pares (campo, valor) para los fields que existen
valuesWithFields :: [Field] -> (Field -> Maybe Value) -> [(Field,
Value)]
valuesWithFields = foldr (\k f -> \g -> case g k of
                                            Nothing -> f g
                                            Just v  -> (k,v) : f g)
                        (const [])

-- Restringe un objeto (representado por una funcion) a solo los fields
especificados.
-- ks es la lista de campos permitidos, f la funcion original.
-- Si el campo que llega esta en f, entonces f k, sino devuelve Nothing
only :: [Field] -> (Field -> Maybe Value) -> Field -> Maybe Value
only ks f = \k -> if elem k ks then f k else Nothing

-- Actualiza o agrega un field con un valor en un objeto (representado
por una funcion).
-- k es el campo a actualizar, v es su valor y f es la funcion
original.
-- Si el campo que llega es igual al que hay que actualizar, se develve
el valor. Sino f k.
update :: Field -> Value -> (Field -> Maybe Value) -> Field -> Maybe
Value
update k v f = \k' -> if k == k' then Just v else f k

-- Crea un objeto (representado por una funcion) con un solo campo y
valor.
singleton :: Field -> Value -> Field -> Maybe Value
singleton k v = \k' -> if k == k' then Just v else Nothing
```

```
-- EJERCICIOS 1

-- Devuelve una lista con todos los índices presentes en la estructura,
en cualquier orden.
indices :: Svson i -> [i]
indices = foldSvson [] (\j _ xs ys -> j : xs ++ ys)

-- Indica si un índice dado existe en la estructura. Debe aprovechar la
propiedad de BST para ser eficiente.
belongs :: Ord i => i -> Svson i -> Bool
belongs = flip (foldSvson (const False)
                          (\j _ g1 g2 -> \i' -> if i' == j
                                                  then True
                                                  else if i' < j
                                                    then g1 i'
                                                    else g2 i'))

-- Busca un índice específico y devuelve los valores de los campos
solicitados para ese índice. Si el índice no existe,
-- devuelve una lista vacía.
lookupProjecting :: Ord i => i -> [Field] -> Svson i -> [Value]
lookupProjecting i' = flip (foldSvson (const [])
                                      (\j f g1 g2 -> \ks -> if null ks
                                                              then []
                                                              else if i'
== j

then valuesOf ks f

else if i' < j

then g1 ks

else g2 ks))

-- Actualiza o inserta un campo en el objeto con el índice dado. Si el
índice no existe, crea un nuevo nodo en la posición correcta del BST.
upsert :: Ord i => i -> Field -> Value -> Svson i -> Svson i
upsert i' k v = recSvson (Obj i' (singleton k v) Empty Empty)
                         (\j f sr1 s1 sr2 s2 -> if i' == j
                                                  then Obj j (update k v
f) s1 s2
                                                  else if i' < j
                                                      then Obj j f sr1 s2
                                                      else Obj j f s1
sr2)

-- Realiza la intersección entre dos estructuras, combinando los
objetos que tienen índices en común.
mkObj :: Ord i => Svson i -> Svson i -> Svson i
mkObj s1 s2 = toSvson (intersect (toList s1) (toList s2))
```

```
intersect :: Eq i => [(i, Field -> Maybe Value)] -> [(i, Field -> Maybe
Value)] -> [(i, Field -> Maybe Value)]
intersect = foldr (\(i,fv) g -> \ifvs' -> if null ifvs'
                                          then []
                                          else if elemI i ifvs'
                                              then (i,fv) : g ifvs'
                                              else g ifvs')
                  (const [])

elemI :: Eq i => i -> [(i, Field -> Maybe Value)] -> Bool
elemI = flip (foldr (\(i, fv) g -> \i' -> i' == i || g i')
                  (const False))

toList :: Svson i -> [(i, Field -> Maybe Value)]
toList = foldSvson [] (\i fv ifvs1 ifvs2 -> (i, fv) : ifvs1 ++ ifvs2)

toSvson :: Ord i => [(i, Field -> Maybe Value)] -> Svson i
toSvson = foldr (\(i,fv) s -> insert i fv s)
                Empty
-- toSvson = foldr (uncurry insert) Empty

insert :: Ord i => i -> (Field -> Maybe Value) -> Svson i -> Svson i
insert i fv = recSvson (Obj i fv Empty Empty)
                  (\j f sr1 s1 sr2 s2 -> if i == j
                                          then Obj j f s1 s2
                                          else if i < j
                                                  then Obj j f sr1 s2
                                                  else Obj j f s1
sr2)

-- Devuelve hasta n objetos que cumplan la condición dada, junto con el
número real de objetos encontrados.
-- Debe priorizar los índices menores (recorrido in-order del BST).
takeUpSatisfying :: Ord i => Int -> (i -> Bool) -> Svson i -> (Svson i,
Int)
takeUpSatisfying _ _ Empty = (Empty, 0)
takeUpSatisfying n p (Obj i fv s1 s2) = let (s1', k1) =
takeUpSatisfying n p s1
                                        in if k1 == n
                                          then (s1', k1)
                                          else if p i
                                                then let (s2', k2) =
takeUpSatisfying (n - k1 - 1) p s2
                                                     in (Obj i fv
s1' s2', k1 + 1 + k2)
                                                else let (s2', k2) =
takeUpSatisfying (n - k1) p s2
                                                     in (merge s1'
s2', k1 + k2)
```

```
takeUpSatisfying' :: Ord i => Int -> (i -> Bool) -> Svson i -> (Svson
i, Int)
takeUpSatisfying' n p s = foldSvson
                                (const (Empty, 0))
                                (\i fv sr1 sr2 -> \m -> let (s1', k1) =
sr1 m
                                                        in if k1 == m
                                                            then
(s1', k1)
                                                            else if p
i

then let (s2', k2) = sr2 (m - k1 - 1)

in (Obj i fv s1' s2', k1 + 1 + k2)

else let (s2', k2) = sr2 (m - k1)

in (merge s1' s2', k1 + k2)) s n
-- Notar que sr1 y sr2 :: Int -> (i -> Bool) -> (Svson i, Int), el tipo
cambia porque paso s n al fold

merge :: Ord i => Svson i -> Svson i -> Svson i
merge s1 s2 = toSvson (toList s1 ++ toList s2)


-- EJERCICIO 2
-- Definir foldSvson y recSvson
foldSvson :: b -> (i -> (Field -> Maybe Value) -> b -> b -> b) -> Svson
i -> b
foldSvson z f Empty            = z
foldSvson z f (Obj i' g s1 s2) = f i' g (foldSvson z f s1) (foldSvson z
f s2)

recSvson :: b -> (i -> (Field -> Maybe Value) -> b -> Svson i -> b ->
Svson i -> b) -> Svson i -> b
recSvson z f Empty            = z
recSvson z f (Obj i' g s1 s2) = f i' g (recSvson z f s1) s1 (recSvson z
f s2) s2




esPar :: Int -> Bool
esPar n = n `mod` 2 == 0
```

```
-- EJEMPLO
ejemplo :: Svson Int
ejemplo =
  Obj 15 (singleton "x" (Vint 15))
    (Obj 13 (singleton "x" (Vint 13))
      (Obj 10 (singleton "x" (Vint 10))
        (Obj 5 (singleton "x" (Vint 5))
          (Obj 3 (singleton "x" (Vint 3)) Empty Empty)
          (Obj 6 (singleton "x" (Vint 6)) Empty Empty))
        (Obj 12 (singleton "x" (Vint 12)) Empty Empty))
      (Obj 14 (singleton "x" (Vint 14)) Empty Empty))
    (Obj 20 (singleton "x" (Vint 20)) Empty Empty)

ejemplo2 :: Svson Int
ejemplo2 = Obj 15 (singleton "x" (Vint 15)) Empty (Obj 20 (singleton
"x" (Vint 20)) Empty Empty)

instance Show i => Show (Svson i) where
  show Empty = "Empty"
  show (Obj i _ l r) =
    "Obj " ++ show i ++ " <func> (" ++ show l ++ ") (" ++ show r ++ ")"

-- EJERCICIO 4
-- para todo i. belongs i = elem i . indices
-- Por principio de extensionalidad. ¿para todo s. para todo i. belongs
i s = (elem i . indices) s?
-- Por defincion de (.), es equivalente: ¿para todo s. para todo i.
belongs i s = elem i (indices s)?
-- Sea s' :: Svnson i, i' :: i. Se demouestra por principio de
induccion estructural sobre s':
-- Caso base, s' = Empty
-- ¿belongs i' Empty = elem i' (indices Empty)?
-- Caso inductivo, s' = Obj i f s1 s2
-- HI1) ¡belongs i' s1 = elem i' (indices s1)!
-- HI2) ¡belongs i' s2 = elem i' (indices s2)!
-- TI)  ¿belongs i' (Obj i f s1 s2) = elem i' (indices (Obj i f s1
s2))?

-- Caso base
-- Lado izq                          -- Lado der
  belongs i' Empty                      elem i' (indices Empty)
= ----------------  (belongs.1)     =          -------------
(indices.1)
  False                                 elem i' []
                                    = ----------  (elem.1)
                                      False
-- Como el lado izq y el lado der son iguales, queda demostrado el caso
base

-- Caso ind
```

```
-- Lado izq
  belongs i' (Obj i f s1 s2)
= -------------------------  (belongs.2)
  if i' == i then True else if i' < i then belongs i' s1 else belongs
i' s2
=                                          -------------  (HI1)
  if i' == i then True else if i' < i then elem i' (indices s1) else
belongs i' s2
=
-------------  (HI2)
  if i' == i then True else if i' < i then elem i' (indices s1) else
elem i' (indices s2)

-------------------------------------------------------------------------
---------------

-- Lado der
  elem i' (indices (Obj i f s1 s2))
=          ----------------------  (indices.2)
  elem i' (i : indices s1 ++ indices s2)
= ------------------------------------ (elem.2)
  i' == i || elem i' (indices s1 ++ indices s2)
=           --------------------------------  (LEMA: elem i (xs ++
ys) = elem i xs || elem i ys)
  i' == i || elem i' (indices s1) || elem i' (indices s2)
  ------------------------------------------------------

-- Parto en casos sobre i e i'

-- C1, i' == i
-- Lado izq
  if i' == i then True else if i' < i then elem i' (indices s1) else
elem i' (indices s2)
=
-------------------------------------------------------------------------
--------------- (C1)
  True

-- Lado der
  i' == i || elem i' (indices s1) || elem i' (indices s2)
= -------                                    (C1)
  True || elem i' (indices s1) || elem i' (indices s2)
= ------------------------------------    (True || b = True)
  True
-- Lado izq y lado der son iguales, vale C2

-- C2, i' < i
-- Lado izq
  if i' == i then True else if i' < i then elem i' (indices s1) else
elem i' (indices s2)
```

```
=
-------------------------------------------------------------------------
---------------- (C2)
  elem i' (indices s2)

-- Lado der
  i' == i || elem i' (indices s1) || elem i' (indices s2)
= -------                                    (C2)
  False || elem i' (indices s1) || elem i' (indices s2)
=                                  --------------------   (Por inv. BST
y C2, i' < indices s1)
  False || elem i' (indices s1) || False
=          ----------------------------  (b || False = b)
  False || elem i' (indices s1)
= ----------------------------           (False || b = b)
  elem i' (indices s1)
-- Lado izq y lado der son iguales, vale C2

-- C3, i' > i
-- Lado izq
  if i' == i then True else if i' < i then elem i' (indices s1) else
elem i' (indices s2)
=
-------------------------------------------------------------------------
---------------- (C3)
  elem i' (indices s2)

-- Lado der
  i' == i || elem i' (indices s1) || elem i' (indices s2)
= -------                                                  (C3)
  False || elem i' (indices s1) || elem i' (indices s2)
=          -------------------                             (Por inv. BST
y C2, i' < indices s1)
  False || False || elem i' (indices s2)
=          ----------------------------                    (Flase || b =
b)
  False || elem i' (indices s2)
= ----------------------------                             (Flase || b =
b)
  elem i' (indices s2)
-- Lado izq y lado der son iguales, vale C3

-- Como valen C1, C2 y C3, vale el caso ind.


-- LEMA: elem i (xs ++ ys) = elem i xs || elem i ys
-- Sea i' :: a, xs', ys' :: [a]. Se demostrara por principio de
induccion estructural sobre xs':
-- Caso base, xs' = []
-- ¿elem i' ([] ++ ys') = elem i' [] || elem i' ys'?
-- Caso ind, xs' = x:xs
```

```
-- HI) ¡elem i' (xs ++ ys') = elem i' xs || elem i' ys'!
-- TI) ¿elem i' ((x:xs) ++ ys') = elem i' (x:xs) || elem i' ys'?

-- Caso base
-- Lado izq                    -- Lado der
  elem i' ([] ++ ys')             elem i' [] || elem i' ys'
=          --------- (++.1)   = ----------                (elem.1)
  elem i' ys'                     Flase || elem i' ys'
                                = -------------------      (False || b =
b)
                                  elem i' ys'
-- Lado izq y lado der son iguales, vale el caso base

-- Caso ind
-- Lado izq
  elem i' ((x:xs) ++ ys')
=          -------------  (++.2)
  elem i' (x : (xs ++ ys'))
= ------------------------  (elem.2)
  i' == x || elem i' (xs ++ ys')
=            ------------------ (HI)
  i' == x || elem i' xs || elem i' ys'

-- Lado der
  elem i' (x:xs) || elem i' ys'
= --------------               (elem.2)
  (i' == x || elem i' xs) || elem i' ys'
= ------------------------------------ (asociatividad del ||)
  i' == x || elem i' xs || elem i' ys'
-- Lado izq y lado der son iguales, vale el caso ind
```

SLICE:

```
data SliceExp a = Base [a]
                | Take Int (SliceExp a)
                | Drop Int (SliceExp a) deriving (Show)

ej :: SliceExp Int
ej = Take 3 (Take 2 (Drop 3 (Drop 1 (Base [1..10]))))


-- EJERCICIO 1
-- Definir por recursion explicita las siguientes funciones:

-- a. Devuelve la lista final, luego de evaluar todas las operaciones.
materialize :: SliceExp a -> [a]

-- b. Devuelve la longitud final de un SliceExp.
lenS :: SliceExp a -> Int

-- c. Simplifica expresiones de manera tal que no hay dos Take ni dos
Drop seguidos, y solo numeros positivos.
normalize :: SliceExp a -> SliceExp a

-- d. Aplica take a una expresión SliceExp ya normalizada, sin
evaluarla completamente. La expresion que devuelve
-- no inicia con el contrsuctor Take.
takeS :: Int -> SliceExp a -> SliceExp a


-- EJERCICIO 2
-- Demostrar: lenS . normalize = lenS


-- EJERCICIO 3
-- Definir el esquema primitivo y recursivo de SliceExp a.


-- EJERCICIO 4
-- Escribir las versiones del ejecicio 1 sin ulizar recursion
explicita.
```

libre

PRACTICA SINCRONICA:

```
data Fila a = Fin
            | Celda Int a (Fila a)
--sawady.faso@gmail.com

-- f Fin = ...
-- f (Celda i a fil) = .. f fil

-- cuenta la cantidad de veces que aparece un elemento que cumple el
predicado
countF :: (a -> Bool) -> Fila a -> Int
countF p Fin             = 0
countF p (Celda i a fil) = fromEnum (p a) + (countF p fil)

-- le suma N a los elementos donde el predicado da verdadero
sumarN :: (a -> Bool) -> Int -> Fila a -> Fila a
sumarN f n Fin             = Fin
sumarN f n (Celda i a fil) = Celda (sumarSiCumple f a i n) a (sumarN f
n fil)


sumarSiCumple:: (a -> Bool) -> a -> Int -> Int  -> Int
sumarSiCumple f a i n = if f a then i+n else i


-- Junta dos filas manteniendo el orden de los elementos
concatenarF :: Fila a -> Fila a -> Fila a
concatenarF Fin             f2 = f2
concatenarF (Celda i a fil) f2 = Celda i a (concatenarF fil f2)


-- transforma cada elemento aplicando una función a los mismos
mapF :: (a -> b) -> Fila a -> Fila b
mapF f Fin             = Fin
mapF f (Celda i a fil) = Celda i (f a) (mapF f fil)

-- transforma una fila de filas en una fila
aplanar :: Fila (Fila a) -> Fila a


-- los elementos iguales los colapsa en una misma posición (sean
contiguos o no)
comprimir :: Fila a -> Fila a

-- denota la composición de las funciones manteniendo el orden de
aparición y aplicandolas las veces que aparezca
componer :: Fila (a -> a) -> (a -> a)
componer Fin             = id
componer (Celda i f fil) = many i f . (componer fil)
```

```
data Functions a = Id
                 | F (a -> a) (Functions a)
                 | B (a -> Bool) (Functions a) (Functions a)

evalF :: Functions a -> (a -> a)
manyF :: Int -> (a -> a) -> Functions a
toFila :: Functions a -> Fila a



data Pizza = Prepizza | Capa Ingrediente Pizza

data Ingrediente = Aceitunas Int | Jamón | Queso | Salsa | MezclaRara

f Prepizza = ...
f (Capa i pz) = ... f pz

cantidadDeCapas :: Pizza -> Int
cantidadDeCapas Prepizza    = 0
cantidadDeCapas (Capa i pz) = 1 + cantidadDeCapas pz

cantidadDeAceitunas :: Pizza -> Int
cantidadDeAceitunas Prepizza    = 0
cantidadDeAceitunas (Capa i pz) = aceitunas i + cantidadDeAceitunas pz

aceitunas (Aceitunas n) = n
aceitunas _             = 0

duplicarAceitunas :: Pizza -> Pizza
duplicarAceitunas Prepizza    = Prepizza
duplicarAceitunas (Capa i pz) = Capa (duplicarSiAceitunas i)
(duplicarAceitunas pz)

duplicarSiAceitunas (Aceitunas n) = Aceitunas (n*2)
duplicarSiAceitunas i             = i

sinLactosa :: Pizza -> Pizza
sinLactosa Prepizza    = Prepizza
sinLactosa (Capa i pz) =
    if tieneLactosa i
       then sinLactosa pz
       else Capa i (sinLactosa pz)

tieneLactosa Queso = True
tieneLactosa _     = False

aptaIntolerantesLactosa :: Pizza -> Bool
aptaIntolerantesLactosa Prepizza    = True
aptaIntolerantesLactosa (Capa i pz) = not (tieneLactosa i) &&
aptaIntolerantesLactosa pz
```

```
conDescripcionMejorada :: Pizza -> Pizza
conDescripcionMejorada Prepizza    = Prepizza
conDescripcionMejorada (Capa i pz) = juntarAceitunas i
(conDescripcionMejorada pz)

juntarAceitunas (Aceitunas n) (Capa (Aceitunas m) pz) = Capa (Aceituna
(n+m)) pz
juntarAceitunas i             pz                      = Capa i pz
```

libre

MULTISET:

```
-- Los multisets son una estructura de datos que permiten saber cuántas
veces fue ingresado un elemento.
-- Para expresar sus operaciones, utilizaremos el lenguaje expresado
por el tipo MSExp.

data MSExp a
    = EmptyMS                       -- un multiset vacío
    | AddMS a (MSExp a)             -- agrega una ocurrencia del elemento
    | RemoveMS a (MSExp a)          -- quita una ocurrencia del elemento
(si hay alguna)
    | UnionMS (MSExp a) (MSExp a) -- combina dos multisets, sumando
ocurrencias
    | MapMS (a -> a) (MSExp a)    -- transforma todos los elementos con
una función


-- EJERCICIO 1
-- Definir por recursión explícita las siguientes funciones

-- a. Describe la cantidad de ocurrencias del elemento dado en el
multiset.
-- AYUDA 1: como la función pedida NO se puede hacer por recursión
estructural, considerar definir como función auxiliar
-- occursMESWith :: a-> (a->a)-> MSExp a-> Int por recursión
estructural e invocarla con los argumentos correspondientes
-- AYUDA 2: el número retornado NO puede ser negativo. O sea, los
RemoveMS que no encuentran un AddMS correspondiente no deben tener
efecto.
occursMSE :: Eq a => a -> MSExp a -> Int
occursMSE e ms = occursMESWith' e id ms

occursMESWith :: Eq a => a -> (a -> a) -> MSExp a -> Int
occursMESWith e f EmptyMS           = 0
occursMESWith e f (AddMS x ms)      = unoSi (e == f x) + occursMESWith
e f ms
occursMESWith e f (RemoveMS x ms)   = if occursMESWith e f ms > 0 then
occursMESWith e f ms - unoSi (e == f x) else occursMESWith e f ms
occursMESWith e f (UnionMS ms1 ms2) = occursMESWith e f ms1 +
occursMESWith e f ms2
occursMESWith e f (MapMS g ms)      = occursMESWith e (f . g) ms

unoSi:: Bool -> Int
unoSi True  = 1
unoSi False = 0

occursMESWith' :: Eq a => a -> (a -> a) -> MSExp a -> Int
occursMESWith' e f ms = (foldMS (\_ _ -> 0) -- const(const 0)
                                (\x n e' f' -> unoSi (e' == f' x) + n
e' f')
```

```
                                (\x n e' f' -> if n e' f' > 0 then n e'
f' - unoSi (e' == f' x) else n e' f')
                                (\n m e' f' -> n e' f' + m e' f')
                                (\g n e' f' -> n e' (f' . g))
                        ) ms e f

-- b. Describe el multiset resultante de eniminar todas las ocurrencias
de los elementos que no cumplen con el predicado dado.
-- AYUDA: tener en cuenta que en el caso de MapMS, el elemento que se
debe analizar es el procesado por la función dada en ese MapMS,
-- y no el elemento sin procesar.
filterMSE :: (a -> Bool) -> MSExp a -> MSExp a
filterMSE p EmptyMS           = EmptyMS
filterMSE p (AddMS x ms)      = if p x then AddMS x (filterMSE p ms)
else (filterMSE p ms)
filterMSE p (RemoveMS x ms)   = RemoveMS x (filterMSE p ms)
filterMSE p (UnionMS ms1 ms2) = UnionMS (filterMSE p ms1) (filterMSE p
ms2)
filterMSE p (MapMS f ms)      = MapMS f (filterMSE (p . f) ms)

filterMSE' :: (a -> Bool) -> MSExp a -> MSExp a
filterMSE' = flip (foldMS   (const EmptyMS)
                            (\x ms p    -> if p x then AddMS x (ms p)
else ms p)
                            (\x ms p    -> RemoveMS x (ms p))
                            (\ms1 ms2 p -> UnionMS (ms1 p) (ms2 p))
                            (\f ms p    -> MapMS f (ms (p . f))))

-- c. Indica si la expresión de multiset dada es válida.
-- Una expresión de multiset es inválida si tiene más RemoveMS que
AddMS para un elemento determinado; en el caso de una UnionMS,
-- cada parte se considera por separado.
isValidMSE :: Eq a => MSExp a -> Bool
isValidMSE = isValidMSEWith id

-- Función auxiliar con transformación acumulada (para MapMS)
isValidMSEWith :: Eq a => (a -> a) -> MSExp a -> Bool
isValidMSEWith f EmptyMS           = True
isValidMSEWith f (AddMS x ms)      = isValidMSEWith f ms
isValidMSEWith f (RemoveMS x ms)   = occursMESWith (f x) f ms > 0 &&
isValidMSEWith f ms
isValidMSEWith f (UnionMS ms1 ms2) = isValidMSEWith f ms1 &&
isValidMSEWith f ms2
isValidMSEWith f (MapMS g ms)      = isValidMSEWith (f . g) ms

isValidMSEWith' :: Eq a => (a -> a) -> MSExp a -> Bool
isValidMSEWith' = flip (recMS   (const True)
                                (\x b _     f -> b f)
                                (\x b ms    f -> occursMESWith (f x) f
ms > 0 && b f)
                                (\b1 _ b2 _ f -> b1 f && b2 f)
```

```
                                (\g b _     f -> b (f . g)))

-- d. Describe la lista de todas las ocurrencias de los elementos del
multiset dado.
evalMSE :: Eq a => MSExp a -> [a]
evalMSE EmptyMS           = []
evalMSE (AddMS x ms)      = x : evalMSE ms
evalMSE (RemoveMS x ms)   = remove x (evalMSE ms)
evalMSE (UnionMS ms1 ms2) = evalMSE ms1 ++ evalMSE ms2
evalMSE (MapMS f ms)      = map f (evalMSE ms)

remove :: Eq a => a -> [a] -> [a]
remove x []     = []
remove x (y:ys) = if x == y then ys else y : remove x ys

evalMSE' :: Eq  a => MSExp a -> [a]
evalMSE' = foldMS [] (:) remove (++) map

remove' :: Eq a => a -> [a] -> [a]
remove' = flip (recr    (const [])
                        (\y ys zs x -> if x == y then ys else y : zs
x))

recr :: b -> (a -> [a] -> b -> b) -> [a] -> b
recr z f []     = z
recr z f (x:xs) = f x xs (recr z f xs)

-- Suponiendo que recibe una expresión de multiset válida y describe al
multiset resultante de simplificar el dado,
-- aplicando las siguientes reglas en todos los lugares posibles:
-- UnionMS EmptyMS e      ⇒ e
-- UnionMS e EmptyMS      ⇒ e
-- MapMS f EmptyMS        ⇒ EmptyMS
-- RemoveMS x (AddMS x e) ⇒ e
simpMSE' ::  Eq a => MSExp a -> MSExp a
simpMSE' = foldMS EmptyMS AddMS simpRemoveMS simpUnionMS simpMapMS

simpRemoveMS :: Eq a => a -> MSExp a -> MSExp a
simpRemoveMS x (AddMS y ms) = if x == y then ms else RemoveMS x (AddMS
y ms)
simpRemoveMS x ms           = RemoveMS x ms

simpUnionMS :: MSExp a -> MSExp a -> MSExp a
simpUnionMS ms1 EmptyMS = ms1
simpUnionMS EmptyMS ms2 = ms2
simpUnionMS ms1 ms2     = UnionMS ms1 ms2

simpMapMS :: (a -> a) -> MSExp a -> MSExp a
simpMapMS f EmptyMS = EmptyMS
simpMapMS f ms      = MapMS f ms
```

```
-- EJERCICIO 3
foldMS :: b -> (a -> b -> b) -> (a -> b -> b) -> (b -> b -> b) -> ((a
-> a) -> b -> b) -> MSExp a -> b
foldMS e a r u m EmptyMS           = e
foldMS e a r u m (AddMS x ms)      = a x (foldMS e a r u m ms)
foldMS e a r u m (RemoveMS x ms)   = r x (foldMS e a r u m ms)
foldMS e a r u m (UnionMS ms1 ms2) = u (foldMS e a r u m ms1) (foldMS e
a r u m ms2)
foldMS e a r u m (MapMS f ms)      = m f (foldMS e a r u m ms)

recMS :: b -> (a -> b -> MSExp a -> b) -> (a -> b -> MSExp a -> b) ->
(b -> MSExp a -> b -> MSExp a -> b) ->
    ((a -> a) -> b -> MSExp a -> b) -> MSExp a -> b
recMS e a r u m EmptyMS           = e
recMS e a r u m (AddMS x ms)      = a x (recMS e a r u m ms) ms
recMS e a r u m (RemoveMS x ms)   = r x (recMS e a r u m ms) ms
recMS e a r u m (UnionMS ms1 ms2) = u (recMS e a r u m ms1) ms1 (recMS
e a r u m ms2) ms2
recMS e a r u m (MapMS f ms)      = m f (recMS e a r u m ms) ms


-- EJEMPLOS
ejemplo :: MSExp Int
ejemplo =
  MapMS (2*) (
    UnionMS
      (RemoveMS 2 (AddMS 1 (AddMS 2 (AddMS 2 (AddMS 3 (AddMS 3 (AddMS 3
EmptyMS)))))))
      (AddMS 1 (AddMS 4 EmptyMS))
  )


instance Show a => Show (MSExp a) where
  show EmptyMS = "EmptyMS"
  show (AddMS x ms) = "AddMS " ++ show x ++ " (" ++ show ms ++ ")"
  show (RemoveMS x ms) = "RemoveMS " ++ show x ++ " (" ++ show ms ++
")"
  show (UnionMS ms1 ms2) = "UnionMS (" ++ show ms1 ++ ") (" ++ show ms2
++ ")"
  show (MapMS _ ms) = "MapMS <func> (" ++ show ms ++ ")"

-- EJERCICIO 2
-- Demostrar la sigueinte propiedad: evalMSE . simpMSE = evalMSE
-- Por principio de extensionalidad. ¿para todo ms :: MSExp a. (evalMSE
. simpMSE) ms = evalMSE ms?
-- Por defincion de (.), es quivalente: ¿para todo ms :: MSExp a.
evalMSE (simpMSE ms) = evalMSE ms?
-- Por principio de induccion estructural sobre la estructura de ms':
-- Caso base, ms' = EmptyMS
-- ¿evalMSE (simpMSE EmptyMS) = evalMSE EmptyMS?
```

```
-- Caso ind 1, ms' = AddMS x ms
-- HI1) ¡evalMSE (simpMSE ms) = evalMSE ms!
-- TI1) ¿evalMSE (simpMSE (AddMS x ms)) = evalMSE (AddMS x ms)?
-- Caso ind 2, ms' = RemoveMS x ms
-- HI2) ¡evalMSE (simpMSE ms) = evalMSE ms!
-- TI2) ¿evalMSE (simpMSE (RemoveMS x ms)) = evalMSE (RemoveMS x ms)?
-- Caso ind 3, ms' = UnionMS ms1 ms2
-- HI3.1) ¡evalMSE (simpMSE ms1) = evalMSE ms1!
-- HI3.2) ¡evalMSE (simpMSE ms2) = evalMSE ms2!
-- TI3)   ¿evalMSE (simpMSE (UnionMS ms1 ms2)) = evalMSE (UnionMS ms1
ms2)?
-- Caso ind 4, ms' = MapMS f ms
-- HI4) ¡evalMSE (simpMSE ms) = evalMSE ms!
-- TI4) ¿evalMSE (simpMSE (MapMS f ms)) = evalMSE (MapMS f ms)?

-- Caso base
    evalMSE (simpMSE EmptyMS)
=                                   (simpMSE.1)
    evalMSE EmptyMS
-- Vale el caso base

-- Caso ind 1
    evalMSE (simpMSE (AddMS x ms))
=                                   (simpMSE.2)
    evalMSE (AddMS x (simpMSE ms))
=                                   (evalMSE.2)
    x : (evalMSE (simpMSE ms))
=                                   (HI2)
    x : evalMSE ms
=                                   (evalMSE.2)
    evalMSE (AddMS x ms)
-- Vale el caso ind 1

-- Caso ind 2
    evalMSE (simpMSE (RemoveMS x ms))
=                                   (simpMSE.3)
    evalMSE (simpRemoveMS x (simpMSE ms))
=                                   (LEMA 1)
    remove x (evalMSE (simpMSE ms))
=                                   (HI3)
    remove x (evalMSE ms)
=                                   (evalMSE.3)
    evalMSE (RemoveMS x ms)
-- Vale el caso ind 2

-- Caso ind 3
    evalMSE (simpMSE (UnionMS ms1 ms2))
=                                   (simpMSE.4)
    evalMSE (simpUnionMS (simpMSE ms1) (simpMSE ms2))
=                                   (LEMA 2)
    evalMSE (simpMSE ms1) ++ evalMSE (simpMSE ms2)
```

```
=                                   (HI3.1 y HI3.2)
    evalMSE ms1 ++ evalMSE ms2
=                                   (evalMSE.4)
    evalMSE (UnionMS ms1 ms2)
-- Vale el caso ind 3

-- Caso ind 4
    evalMSE (simpMSE (MapMS f ms))
=                                   (simpMSE.5)
    evalMSE (simpMapMS f (simpMSE ms))
=                                   (LEMA 3)
    map f (evalMSE (simpMSE ms))
=                                   (HI4)
    map f (evalMSE ms)
=                                   (evalMSE.5)
    evalMSE (MapMS f ms)
-- Vale el caso ind 4

-------------------------------------------------------------------------
------------------------

-- LEMA 1:
-- para todo x. para todo ms. evalMSE (simpRemoveMS x ms) = remove x
(evalMSE ms)
-- Sea ms' :: MSExp a, z :: a.
-- Se demostrara por casos sobre ms':
-- Caso 1, ms' = AddMS x ms
-- ¿evalMSE (simpRemoveMS z (AddMS x ms)) = remove z (evalMSE (AddMS x
ms))?
-- Caso 2: ms' /= AddMS x ms.
-- ¿evalMSE (simpRemoveMS z ms') = remove z (evalMSE ms')?

-- Caso 1:
-- LI.
    evalMSE (simpRemoveMS z (AddMS x ms))
=                                   (simpRemoveMS.1)
    evalMSE (if z == x then ms else RemoveMS z (Add x ms))
=

-- LD
    remove z (evalMSE (AddMS x ms))
=                                   (evalMSE.2)
    remove z (x : evalMSE ms)
=                                   (remove.2)
    if z == x then evalMSE ms else x : remove z (evalMSE ms)
=


        -- Parto en casos sobre el if.
        --  Subcaso 1: z == x.
        --  LI.
```

```
            evalMSE (if z == x then ms else RemoveMS z (Add x ms))
        =                                   (C1)
            evalMSE ms
        -- LD.
            if z == x then evalMSE ms else x : remove z (evalMSE ms)
        =                                   (C1)
            evalMSE ms
        -- Vale C1

        -- Subcaso : z /= x
        -- LI
            evalMSE (if z == x then ms else RemoveMS z (Add x ms))
        =                                    (z /= x)
            evalMSE (RemoveMS z (Add x ms))
        =                                    (evalMSE.3)
            remove z (evalMSE (Add x ms))
        =                                    (evalMSE.2)
            remove z (x : evalMSE ms)
        =                                    (remove.2)
            if z == x then evalMSE ms else x : remove z (evalMSE ms)
        -- Vale C2


-- Caso 2
    evalMSE (simpRemoveMS z ms')
=                                   (simpRemoveMS.1)
    evalMSE (RemoveMS z ms')
=                                   (evalMSE.3)
    remove z (evalMSE ms')
-- Vale el caso 2

--------------------------------------------------------------------------
------------------------


-- LEMA 2:
-- para todo ms1. para todo ms2. evalMSE (simpUnionMS ms1 ms2) =
evalMSE ms1 ++ evalMSE ms2
-- Sean m1, m2 :: MSExp a.
-- Se demostrara por casos sobre m1 y m2:
-- Caso 1, m1 = EmptyMS, m2 = cualquier caso
-- ¿evalMSE (simpUnionMS EmptyMS m2) = evalMSE EmptyMS ++ evalMSE m2?
-- Caso 2, m1 /= EmptyMS, m2 = EmptyMS
-- ¿evalMSE (simpUnionMS m1 EmptyMS) = evalMSE m1 ++ evalMSE EmptyMS?
-- Caso 3, m1 /= EmptyMS, m2 /= EmptevalMSEyMS
-- ¿evalMSE (simpUnionMS m1 m2) =  m1 ++ evalMSE m2?

-- Caso 1
-- LI
    evalMSE (simpUnionMS (simpMSE Empty) (simpMSE m2))
=                               (simpMSE.1)
```

```
    evalMSE (simpUnionMS Empty m2)
=                                 (simpUnionMS.1)
    evalMSE m2
-- LD
    evalMSE Empty ++ evalMSE m2
=                                 (evalMSE.1)
    [] ++ evalMSE m2
=                                 (++.1)
    evalMSE m2
-- Vale el caso 1

-- Caso 2
-- Igual que el caso 1 pero con los parametros invertidos
-- Vale el caso 2

-- Caso 3
-- LI
    evalMSE (simpUnionMS m1 m2)
=                                 (simpUnionMS.3)
    evalMSE (UnionMS m1 m2)
=
    evalMSE m1 ++ evalMSE m2
-- Vale el caso 3

-------------------------------------------------------------------------
------------------------

-- LEMA 3:
-- para todo f. para todo ms. evalMSE (simpMapMS f ms) = map f (evalMSE
ms)
-- Sea ms' :: MSExp a, f :: a -> a.
-- Se demostrara por casos sobre ms':
-- Caso 1, ms' = EmptyMS
-- ¿evalMSE (simpMapMS f EmtyMS) = map f (evalMSE EmptyMS)?
-- Caso 2: ms' /= EmptyMS.
-- ¿evalMSE (simpMapMS f ms') = map f (evalMSE ms')?

-- Caso 1
-- LI
    evalMSE (simpMapMS f EmtyMS)
=                         (simpMapMS.1)
    evalMSE EmptyMS
=                         (evalMSE.1)
    []
-- LD
    map f (evalMSE EmptyMS)
=                         (evalMSE.1)
    map f []
=                         (map.1)
-- Vale el caso 1
```

```
-- Caso 2
    evalMSE (simpMapMS f ms')
=                        (simpMapMS)
    evalMSE (MapMS f ms')
=                        (evalMSE.5)
    map f (evalMSE ms')
-- Vale el caso 2
```

libre

MULTISET LIGHT:

```
data N = Z | S N
-- Z = 0
-- S (S ( S N)) = 3

data MSExp a =  Empty |
                Entry N a | -- n es la cantidad de apariciones de a
                AddAll N (MSExp a) | -- agrega n apariciones
                Union (MSExp a) (MSExp a) -- une las apariciones de los
elementos del primero con los del segundo


-- EJERCICIO 1
-- Definir por recursion explicita las siguientes funciones:

-- Retorna el significado del multiset dado.
evalMSE :: Eq a => MSExp a -> a -> N
evalMSE Empty           y = Z
evalMSE (Entry n x)     y = if x == y then n else Z
evalMSE (AddAll n ms)   y = addN n (evalMSE ms y)
evalMSE (Union ms1 ms2) y = addN (evalMSE ms1 y) (evalMSE ms2 y)

addN :: N -> N -> N
addN Z     m = m
addN (S n) m = S (addN n m)


-- Dada una funcion de actualziacion de numeros naturales y un
multioset naturalizado, describe el multiset que resulta de actualizar
-- todas las ocurrencias de los elementos actuales segund  la funcion
dada.
-- Por ejemplo, si la funcion incrementa el natural en 1, es
equivalente a que todos los elementos existentes incorporan una
ocurrencia mas.
-- {{a, a, b}} = {{a, a, a, b, b}}
updateOcurrsMSE :: (N -> N) -> MSExp a -> MSExp a
updateOcurrsMSE f Empty           = Empty
updateOcurrsMSE f (Entry n x)     = Entry (f n) x
updateOcurrsMSE f (Union ms1 ms2) = Union (updateOcurrsMSE f ms1)
(updateOcurrsMSE f ms2)


-- Describe el resultado de la normalizacion del multiset dado.
-- Un multiset esta normalizado si no contiene ocurrencias del
contrstructor AddAll,
-- y si ningun constructor Union tiene como argumento un constructor
Empty.
normMSE :: MSExp a -> MSExp a
normMSE Empty           = Empty
normMSE (Entry n x)     = Entry n x
normMSE (AddAll n ms)   = updateOcurrsMSE (addN n) (normMSE ms)
```

```
--                       = normAddAll n (normMSE ms)
normMSE (Union ms1 ms2) = normUnion (normMSE ms1) (normMSE ms2)

-- normAddAll :: N -> MSExp a -> MSExp a
-- normAddAll m Empty           = Empty
-- normAddAll m (Entry n x)     = Entry (addN m n) x
-- normAddAll m (Union ms1 ms2) = Union (normAddAll f ms1) (normAddAll
f ms2)

normUnion :: MSExp a -> MSExp a -> MSExp a
normUnion Empty ms2 = ms2
normUnion ms1 Empty = ms1
normUnion ms1 ms2   = Union ms1 ms2


-- Describe la cantidad de "fallas" que tiene el multiset dado para
estar normalziado,
-- (o sea, la cantidad de constructores AddAll ademas de la cantidad de
constructores
-- Empty que son argumento de una Union). Observar que si cantFallas m
= 0, entonces m esta normalizado.
cantFallas :: MSExp a -> Int
cantFallas Empty           = 0
cantFallas (Entry n x)     = 0
cantFallas (AddAll n ms)   = 1 + cantFallas ms
cantFallas (Union ms1 ms2) = unoSiHayEmpty ms1 ms2 + cantFallas ms1 +
cantFallas ms2

unoSiHayEmpty :: MSExp a -> MSExp a -> Int
unoSiHayEmpty Empty _ = 1
unoSiHayEmpty _ Empty = 1
unoSiHayEmpty _ _     = 0


-- EJERCICIO 2
foldMSE :: b -> (N -> a -> b) -> (N -> b -> b) -> (b -> b -> b) ->
MSExp a -> b
foldMSE z e a u Empty           = z
foldMSE z e a u (Entry n x)     = e n x
foldMSE z e a u (AddAll n ms)   = a n (foldMSE z e a u ms)
foldMSE z e a u (Union ms1 ms2) = u (foldMSE z e a u ms1) (foldMSE z e
a u ms2)

recMSE :: b -> (N -> a -> b) -> (N -> b -> MSExp a -> b) -> (b -> MSExp
a ->  b -> MSExp a -> b) -> MSExp a -> b
recMSE z e a u Empty           = z
recMSE z e a u (Entry n x)     = e n x
recMSE z e a u (AddAll n ms)   = a n (recMSE z e a u ms) ms
recMSE z e a u (Union ms1 ms2) = u (recMSE z e a u ms1) ms1 (recMSE z e
a u ms2) ms2
```

```
-- EJERCICIO 3
-- Escribir las versiones del ejecicio 1 sin ulizar recursion
explicita.

evalMSE' :: Eq a => MSExp a -> a -> N
evalMSE' = foldMSE  (const Z)
                    (\n x -> \y -> if x == y then n else Z)
                    (\n f -> \y -> addN n (f y))
                    (\f g -> \y -> addN (f y) (g y))

updateOcurrsMSE' :: (N -> N) -> MSExp a -> MSExp a
updateOcurrsMSE' = flip (foldMSE    (const Empty)
                                    (\n x -> \f -> Entry (f n) x)
                                    (\_ _ -> \_ -> error "...")
                                    (\g h -> \f -> Union (g f) (h f)))

normMSE' :: MSExp a -> MSExp a
normMSE' = foldMSE Empty Entry (updateOcurrsMSE . addN) normUnion

cantFallas' :: MSExp a -> Int
cantFallas' = recMSE 0
                    (\_ _ -> 0)
                    (\_ m _ -> 1 + m)
                    (\n1 ms1 n2 ms2 -> unoSiHayEmpty ms1 ms2 + n1 + n2)
-- EJERCICIO 2
-- Demostrar la siguiente propiedad: cantFallas . normMSE = const 0

-- Por principio de extensionalidad es equivalente a ¿para todo ms ::
MSExp a. (cantFallas . normMSE) ms = const 0 ms?
-- Por definicion de (.), es equivalente a ¿para todo ms. cantFallas
(normMSE ms) = const 0 ms?
-- Por definicion de const, es equivalente: ¿para todo ms. cantFallas
(normMSE ms) = 0?
-- Sea ms' :: MSExp a. Por principio de inducción estructural sobre la
la estructura de ms':
-- Caso base 1, ms' = Empty
-- ¿cantFallas (normMSE Empty) = 0?
-- Caso base 2, ms' = Entry n x
-- ¿cantFallas (normMSE (Entry n x)) = 0?
-- Caso ind 1, ms' = AddAll n ms
-- HI1) ¡cantFallas (normMSE ms) = 0!
-- TI1) ¿cantFallas (normMSE (AddAll n ms)) = 0?
-- ¿cantFallas (normMSE (Entry n x)) = 0?
-- Caso ind 2, ms' = Union ms1 ms2
-- HI1) ¡cantFallas (normMSE ms1) = 0!
-- HI1) ¡cantFallas (normMSE ms2) = 0!
-- TI1) ¿cantFallas (normMSE (Union ms1 ms2)) = 0?

-- Caso base 1
    cantFallas (normMSE Empty)
```

```
=                                (normMSE.1)
    cantFallas Empty
=                                (cantFallas.1)
    0
-- Vale el caso base 1

-- Caso base 2
    cantFallas (normMSE (Entry n x))
=                                (normMSE.2)
    cantFallas (Entry n x)
=                                (cantFallas.2)
    0
-- Vale el caso base 2

-- Caso ind 1
    cantFallas (normMSE (AddAll n ms))
=                                (normMSE.3)
    cantFallas (updateOcurrsMSE (addN n) (normMSE ms))
=                                (LEMA 1)
    cantFallas (normMSE ms)
=                                (HI1)
    0
-- Vale el caso ind 1

-- Caso ind 2
    cantFallas (normMSE (Union ms1 ms2))
=                                (normMSE.4)
    cantFallas (normUnion (normMSE ms1) (normMSE ms2))
=                                (LEMA 2)
    cantFallas (normMSE ms1) + cantFallas (normMSE ms2)
=                                (HI1 y HI2)
    0 + 0
=                                (aritm)
    0
-- Vale el caso ind 2


-- LEMA 1:
-- ¿para todo ms :: MESxp a normalizado. para todo f :: N -> N.
cantFallas (updateOcurrsMSE f ms) = cantFallas ms?
-- Sea ms' :: MSExp a normalizado, f' :: N -> N. Por principio de
induccion estructural sobre la estructura de ms':
-- Caso base 1, ms' = Empty
-- ¿cantFallas (updateOcurrsMSE f Empty) = cantFallas Empty?
-- Caso base 2, ms' = Entry n x
-- ¿cantFallas (updateOcurrsMSE f (Entry n x)) = cantFallas (Entry n
x)?
-- Caso ind 1, ms' = Union ms1 ms2
-- HI2) ¡cantFallas (updateOcurrsMSE f ms1) = cantFallas ms1!
-- HI2) ¡cantFallas (updateOcurrsMSE f ms2) = cantFallas ms2!
```

```
-- TI)  ¿cantFallas (updateOcurrsMSE f (Union ms1 ms2)) = cantFallas
(Union ms1 ms2)?

-- Caso base 1
    cantFallas (updateOcurrsMSE f Empty)
=                                 (updateOcurrsMSE.1)
    cantFallas Empty
-- Vale el caso base 1

-- Caso base 2
-- LI
    cantFallas (updateOcurrsMSE f (Entry n x))
=                                 (updateOcurrsMSE.2)
    cantFallas (Entry (f n) x)
=                                 (cantFallas.2)
    0
-- LD
    cantFallas (Entry n x)
=                                 (cantFallas.2)
    0
-- Vale el caso base 2

-- Caso ind 3
-- LI
    cantFallas (updateOcurrsMSE f (Union ms1 ms2))
=                                 (updateOcurrsMSE.3)
    cantFallas (Union (updateOcurrsMSE f ms1) (updateOcurrsMSE f ms2))
=                                 (cantFallas.4)
    unoSiHayEmpty ms1 ms2 + cantFallas (updateOcurrsMSE f ms1) +
cantFallas (updateOcurrsMSE f ms1)
=                                 (HI1 y HI2)
    unoSiHayEmpty ms1 ms2 + cantFallas ms1 + cantFallas ms2
-- LD
    cantFallas (Union ms1 ms2)
=                                 (cantFallas.4)
    unoSiHayEmpty ms1 ms2 + cantFallas ms1 + cantFallas ms2


-- LEMA 2: cantFallas (normUnion ms1 md2) = cantFallas ms1 + cantFallas
ms2
-- ¿para todo ms1, ms2 :: MESxp a normalizado. cantFallas (normUnion
ms1 md2) = cantFallas ms1 + cantFallas ms2?
-- Sean ms1', ms2' :: MSExp a, normalizados. Se demostará por casos
sobre ms1' y ms2':
-- Caso 1, ms1' = Empty y ms2' = cualquier caso
-- ¿cantFallas (normUnion Empty md2') = cantFallas Empty + cantFallas
ms2'?
-- Caso 2, ms1' != Empty y ms2' = Empty
-- ¿cantFallas (normUnion md1' Empty) = cantFallas ms1' + cantFallas
Empty?
-- Caso 3, ms1' != Empty y ms2' != Empty
```

```
-- ¿cantFallas (normUnion ms1' md2') = cantFallas ms1' + cantFallas
ms2'?

-- Caso 1
-- LI
    cantFallas (normUnion Empty md2')
=                                (normUnion.1)
    cantFallas ms2'
-- LD
    cantFallas Empty + cantFallas ms2'
=                                (cantFallas.1)
    0 + cantFallas ms2'
=                                (aritm)
    cantFallas' ms2'
-- Vale el caso 1

-- Caso 2
-- LI
    cantFallas (normUnion md1' Empty)
=                                (normUnion.2)
    cantFallas ms1'
-- LD
    cantFallas ms1' + cantFallas Empty
=                                (cantFallas.1)
    cantFallas ms1' + 0
=                                (aritm)
    cantFallas' ms1'

-- Caso 3
-- LI
    cantFallas (normUnion ms1' md2')
=                                (normUnion.3)
    cantFallas (Union ms1' ms2')
=                                (cantFallas.4)
    unoSiHayEmpty ms1' ms2' + cantFallas ms1' + cantFallas ms2'
=                                (ms1' y ms2' no son Empty)
    0 + cantFallas ms1' + cantFallas ms2'
=                                (aritm)
    cantFallas ms1' + cantFallas ms2'
-- LD
    cantFallas ms1' + cantFallas ms2'
-- Vale el caso 3
```

MAPA:

```
data Dir = LeftM | RightM | StraightM
data Mapa a = Cofre [a] | Nada (Mapa a) | Bifurcacion [a] (Mapa a)
(Mapa a)

objects :: Mapa a -> [a]
objects (Cofre xs)             = xs
objects (Nada m)               = objects m
objects (Bifurcacion xs ml mr) = xs ++ (objects ml) ++ (objects mr)

mapMp :: (a -> b) -> Mapa a -> Mapa b
mapMp f (Cofre xs) = Cofre (map f xs)
mapMp f (Nada m)   = mapMp f m
mapMp f (Bifurcacion xs ml mr) = Bifurcacion (map f xs) (mapMp f ml)
(mapMp f mr)

hasObjectAt :: (a -> Bool) -> Mapa a -> [Dir] -> Bool
hasObjectAt f (Cofre xs) [] = any f xs
hasObjectAt f (Nada m)  (StraightM:xs) = hasObjectAt f m xs
hasObjectAt f (Bifurcacion xs ml mr) [] = any f xs
hasObjectAt f (Bifurcacion xs ml mr) (LeftM:xs) = hasObjectAt f ml xs
hasObjectAt f (Bifurcacion xs ml mr) (RightM:xs) = hasObjectAt f mr xs
hasObjectAt _ _ _ = False

longestPath :: Mapa a -> [Dir]
longestPath (Cofre xs) = []
longestPath (Nada m)   = StraightM : (longestPath m)
longestPath (Bifurcacion xs ml mr) =
    let (lPath, rPath) = ((longestPath ml), longestPath mr) in
    if length lPath > length rPath
        then LeftM : lPath
        else RightM : rPath

heightM :: Mapa a -> Int
heightM (Cofre xs) = 1
heightM (Nada m)   = 1 + heightM m
heightM (Bifurcacion xs ml mr) = 1 + (max (heightM ml) (heightM mr))

allPaths :: Mapa a -> [[Dir]]
allPaths (Cofre xs) = [[]]
allPaths (Nada m)   = map (StraightM:) (allPaths m)
allPaths (Bifurcacion xs ml mr) = (map (LeftM:) (allPaths ml)) ++ (map
(RightM:) (allPaths mr))

-- Tipar y definir foldM y recM
foldM :: ([a] -> b)
      -> (b -> b)
      -> ([a] -> b -> b -> b)
      -> Mapa a
      -> b
foldM c n b (Cofre xs) = c xs
```

```
foldM c n b (Nada m)   = n (foldM c n b m)
foldM c n b (Bifurcacion xs ml mr) = b xs (foldM c n b ml) (foldM c n b
mr)

recM :: ([a] -> b)
     -> (b -> Mapa a -> b)
     -> ([a] -> b -> b -> Mapa a -> Mapa a -> b)
     -> Mapa a
     -> b
recM c n b (Cofre xs) = c xs
recM c n b (Nada m)   = n (recM c n b m) m
recM c n b (Bifurcacion xs ml mr) = b xs (recM c n b ml) (recM c n b
mr) ml mr

-- Redefinir funciones utilizando foldM o recM
objects' :: Mapa a -> [a]
objects' = foldM id
                 id
                 ((++) . (++))

mapMp' :: (a -> b) -> Mapa a -> Mapa b
mapMp' = foldM (Cofre . map f)
```

FUNNEL:

```
-- Tupla de elementos que cumplen o no un criterio respectivamente
type Partition a = ([a], [a])
-- Un criterio que, de cumplirse un predicado, aplica la primer funcion
o la segunda en caso contrario
data Criteria a b = C (a -> Bool) (a -> b) (a -> b)
-- Una estructura linel no vacia que representa los criterios a
utilizar que se aplican desde el ultimo al primero:
-- Ej: Step c3 (Step c2 (Initial c1)) aplica primero c1, luego c2, por
ultimo c3
data Funnel a b = Initial (Criteria a b) | Step (Criteria a b) (Funnel
a b)


-- Devuelve una tupla cuya primera componente es partition, compuesto
por los elementos que cumplen y no cumplen el criterio dado,
-- y la segunda componente son los elementos que cumplen luego de
aplicarse la funcion del criterio.
partition :: Criteria a b -> [a] -> (Partition a, [b]) -- (([a], [a]),
[b])
partition (C p f g) []     = (([],[]), [])
partition (C p f g) (x:xs) = let ((ts, fs), ys) = partition (C p f g)
xs -- El resultado de la recursion
                                in if p x -- si se cumple, proceso el
primer elemento
                                    then ((x:ts, fs), f x : ys)
                                    else ((ts, x:fs), g x : ys)

-- Aplica el nuevo criterio a los elementos que no cumplieron criterios
anteriores (la parte de la partición que quedó sin procesar),
-- y luego combina los resultados nuevos con los previos usando la
función combinadora.
step :: Criteria a b -> ([b] -> b) -> (Partition a, [b]) -> (Partition
a, [b])
step c f ((ts, fs), ys) = let ((ts', fs'), ys') = partition c fs -- los
elementos de fs procesados con el nuevo criterio
                          in ((ts ++ ts', fs'), f ys' : ys)

-- Para un elemento a, primero evalúa el primer criterio y luego, sobre
el resultado b obtenido,
-- aplica el segundo criterio para finalmente obtener c.
composeC :: Criteria a b -> Criteria b c -> Criteria a c
composeC (C p1 f1 g1) (C p2 f2 g2) = C (\x -> p1 x && p2 (f1 x)) (f2 .
f1) (g2 . g1)


-- EJERCICIO 1
-- Definir por recursion explicita las siguientes funciones:

-- a. Dado un funnel, una función que "reduce" (foldr) una lista de
resultados, y una lista de tipo [a],
```

```
-- retorna la particion de elementos a tras aplicar el funnel.
appF :: Funnel a b -> ([b] -> b) -> [a] -> (Partition a, [b])

-- b. Computa el complemento de un funnel que filtra y transforma
infomacion con los criterios del funnel dado negados.
-- Tener en cuenta que al negra un criterio, los predicados de
transformacion deben intercambiarse adecuadamente.
complementF :: Funnel a b -> Funnel a b

-- c. Dado un funnel, retorna uno donde los criterios se aplican al
reves.
-- Pensar la solucion como una lista
reverseF :: Funnel a b -> Funnel a b

-- d. Dado un funnel y una funcion b -> c, lo retorna mapeando sus
funciones de a -> b por a -> c
mapF :: (b -> c) -> Funnel a b -> Funnel a c

-- e. Une dos funnel, paso a paso. Cada paso del primer funnel alimenta
de datos al segundo, y si hay mas pasos en alguno de los dos,
-- se ignoran.
zipF :: Funnel a b -> Funnel b c -> Funnel a c


-- EJERCICIO 2
-- Definir foldFunnel y recFunnel.


-- EJERCICIO 3
-- Escribir las versiones del ejecicio 1 sin ulizar recursion
explicita.


-- EJERCICIO 4
-- ¿para todo fn. para todo f. para todo xs. appF fn f xs = appF
(complementF (complementF fn)) f xs?
```

Ejercicios 2do parcial UNQ

1. Dar el tipo y escribir el esquema de recursión primitiva de **`TCommand`**.

2. Dar el tipo y escribir la función que expresa el esquema de recursión estructural sobre **`TCommand`**, sin utilizar recursión explícita.

3. Definir sin utilizar recursión explícita la función **`cantSeq :: TCommand -> Int`**, que dado un programa de gráficos de tortuga, describe la cantidad de constructores **`(:#:)`** del mismo.

4. Definir sin utilizar recursión explícita la función **`cantSeqsALaIzqDeSeq :: TCommand -> Int`**, que dado un programa de gráficos de tortuga, describe la cantidad total de constructores **`(:#:)`** del mismo que están en algún argumento izquierdo de un **`(:#:)`**.

5. Definir sin utilizar recursión explícita la función **`assocDer :: TCommand -> TCommand`**, que dado un programa de tortuga, describe uno equivalente que no tiene argumentos a izquierda de un **`(:#:)`** que estén formados por **`(:#:)`**.
   SUGERENCIA: Considerar definir una función auxiliar para el caso inductivo.

6. Definir la función **`scaleTC :: TCommand -> Float -> TCommand`**, *expresada como recursión estructural implícita* sobre **`TCommand`**. Esta función escala cada comando **`Go`** de su argumento según el factor dado. Por ejemplo, **`scaleTC (Go 10 :#: Go 30) 2 = (Go 20 :#: Go 60)`**.

7. Dar una versión de la función **`cantSeqs`** dada a continuación, sin usar recursión explícita

```
cantSeqs (c1 :#: c2) = let (cs1, csis1) = cantSeqs c1
                           (cs2, csis2) = cantSeqs c2
                       in (1+cs1+cs2, cs1+csis1+csis2)
cantSeqs c           = (0,0)
```

8. Demostrar que la versión dada de **`cantSeqs`** en el ejercicio anterior efectivamente lo es.

libre