1. Las listas finitas pueden especificarse como un TAD con los constructores:

   - `Nil`: Construye una lista vacía.
   - `Cons`: Agrega un elemento a la lista.

   y las siguientes operaciones:

   - `null`: Nos dice si la lista es vacía o no.
   - `head`: Devuelve el primer elemento de la lista.
   - `tail`: Devuelve todos los elementos de la lista menos el primero.

   a) Dar una especificación algebraica del TAD listas finitas.

   b) Dar una especificación tomando como modelo las secuencias.

   c) Asumiendo que a es un tipo con igualdad, especificar una función `inL :: List a -> a -> Bool` tal que `inL xs x` $\equiv$ `True` si y solo si `x` es un elemento de `xs`.

   d) Especificar una función que elimina todas las ocurrencias de un elemento dado.

Soluciones

a)

TAD `List (a :: Set)` where
`import Bool`
`Nil :: List a`
`Cons :: a -> List a -> List a`
`null :: List a -> Bool`
`head :: List a -> a`
`tail :: List a -> List a`
`null Nil` $\equiv$ `True`
`null (Cons x xs)` $\equiv$ `False`
`head (Cons x xs)` $\equiv$ `x`
`tail (Cons x xs)` $\equiv$ `xs`

b)

`Nil`	$\equiv \langle\rangle$
`Cons` $(x\ \langle x_1, \ldots, x_n\rangle)$	$\equiv \langle x, x_1, \ldots, x_n\rangle$
`null` $\langle\rangle$	$\equiv$ `True`
`null` $\langle x_1, \ldots, x_n\rangle$	$\equiv$ `False` (si $n > 0$)
`head` $\langle x_1, \ldots, x_n\rangle$	$\equiv x_1$
`tail` $\langle x_1, \ldots, x_n\rangle$	$\equiv \langle x_2, \ldots, x_n\rangle$

c)

```
inL Nil x          ≡ False
inL (Cons x xs) y  ≡ if x == y then True else inL xs y
```

d)

```
delete Nil x          ≡ Nil
delete (Cons x xs) y  ≡ if x == y then delete xs y
                                  else Cons x (delete xs y)
```

2. Dado el TAD pilas, con las siguientes operaciones:

   - `Empty`: Construye una pila inicialmente vacía.
   - `Push`: Agrega un elemento al tope de la pila.
   - `isEmpty`: Devuelve verdadero si su argumento es una pila vacía, falso en caso contrario.
   - `top`: Devuelve el elemento que se encuentra al tope de la pila.
   - `pop`: Saca el elemento que se encuentra al tope de la pila.

   Dar una especificación algebraica del TAD pilas y una especificación tomando como modelo las secuencias.

Soluciones

- TAD `Stack (a :: Set)` where

```
import Bool
Empty :: Stack a
Push :: a -> Stack a -> Stack a
isEmpty :: Stack a -> Bool
top :: Stack a -> a
pop :: Stack a -> Stack a
```

```
isEmpty Empty         ≡ True
isEmpty (Push x xs)   ≡ False
top (Push x xs)       ≡ x
pop (Push x xs)       ≡ xs
```

`Empty`	$\equiv \langle\rangle$
`Push` $(x\ \langle x_1, \ldots, x_n\rangle)$	$\equiv \langle x, x_1, \ldots, x_n\rangle$
`isEmpty` $\langle\rangle$	$\equiv$ `True`
`isEmpty` $\langle x_1, \ldots, x_n\rangle$	$\equiv$ `False` (si $n > 0$)
`top` $\langle x_1, \ldots, x_n\rangle$	$\equiv x_1$
`pop` $\langle x_1, \ldots, x_n\rangle$	$\equiv \langle x_2, \ldots, x_n\rangle$

3. Asumiendo que a es un tipo con igualdad, completar la siguiente especificación algebraica del TAD conjunto.

TAD `Conjunto (a :: Set)` where

```
import Bool
Vacio :: Conjunto a
Insertar :: a -> Conjunto a -> Conjunto a
borrar :: a -> Conjunto a -> Conjunto a
esVacio :: Conjunto a -> Bool
union :: Conjunto a -> Conjunto a -> Conjunto a
interseccion :: Conjunto a -> Conjunto a -> Conjunto a
resta :: Conjunto a -> Conjunto a -> Conjunto a
```

¿Que pasaría si se agregase una función `choose :: Conjunto a -> a`, tal que `choose (insertar x c)` $\equiv$ `x`?

Solución

```
insertar x (insertar y c)        ≡ insertar y (insertar x c)
insertar x (insertar x c)        ≡ insertar x c
borrar x Vacio                   ≡ Vacio
borrar x (insertar y c)          ≡ if x == y
                                   then borrar x c
                                   else insertar y (borrar x c)
esVacio Vacio                    ≡ True
esVacio (insertar x xs)          ≡ False
union c Vacio                    ≡ c
union c (insertar x c')          ≡ union (insertar x c) c'
in x Vacio                       ≡ False
in x (insertar y c)              ≡ if x == y then True
                                             else in x c
interseccion c Vacio             ≡ Vacio
interseccion c (insertar x c')   ≡ if x in c
                                   then insertar x (interseccion c c')
                                   else interseccion c c'
resta c Vacio                    ≡ c
resta c (insertar x c')          ≡ resta (borrar x c) c'
inL (Cons x xs) y                ≡ if x == y then True else inL xs y
```

4. El TAD *priority queue* es una cola en la cual cada elemento tiene asociado un valor que es su prioridad (a dos elementos distintos le corresponden prioridades distintas). Los valores que definen la prioridad de los elementos pertenecen a un conjunto ordenado. Las siguientes son las operaciones soportadas por este TAD:

   - `Vacia`: Construye una priority queue vacía.
   - `Poner`: Agrega un elemento a una priority queue con una prioridad dada.
   - `primero`: Devuelve el elemento con mayor prioridad de una priority queue.
   - `sacar`: Elimina de una priority queue el elemento con mayor prioridad.
   - `esVacia`: Determina si una priority queue es vacía.
   - `union`: Une dos priority queues.

Dar una especificación algebraica del TAD priority queue y una especificación tomando como modelo los conjuntos.

5. Agregar a la siguiente definición del TAD árboles balanceados, una especificación para las operaciones `size` y `expose`:

TAD `BalT (a :: Ordered Set)` where

```
import Maybe, Nat, Tupla2
Empty :: BalT a
join :: BalT a -> Maybe a -> BalT a -> BalT a
size :: BalT a -> Nat
expose :: BalT a -> Maybe (BalT a, a, BalT a)
```

   - La operación `join` toma un árbol `l`, un elemento opcional y un árbol `r`. Si `l` y `r` son árboles de búsqueda balanceados tales que todos los elementos de `l` sean menores que todos los elementos de `r` y el elemento opcional es más grande que los de `l` y menor que los de `r`, entonces `join` crea un nuevo árbol de búsqueda balanceado.
   - Las operaciones `Empty` y `size` son obvias.
   - La operación `expose` toma un árbol `t` y nos da `Nothing` si el árbol está vacío, y en otro caso nos devuelve un árbol izquierdo, un elemento raíz y un árbol derecho de un árbol de búsqueda que contiene todos los elementos de `t`.

   Notar que `join` no es simplemente un constructor sino que tiene que realizar cierto trabajo para devolver un árbol balanceado. Debido a esto es conveniente especificar `expose` por casos sobre su resultado.

6. Demostrar que `(uncurry zip) . unzip` $\equiv$ `id`, siendo:

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys         = []
zip (x:xs) []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

unzip :: [(a,b)] -> ([a], [b])
unzip []         = ([], [])
unzip ((x,y):ps) = (x:xs, y:ys)
                   where (xs,ys) = unzip ps

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