Tema A:

Ejercicio 1:

a) Definir el tipo `Moneda` cuyos constructores son: `Uno`, `Dos`, `Cinco`, `Diez`. Asegurarse que el tipo esté en la clase `Show` y que pertenezca a la clase que nos permita escribir listas al estilo de `[Uno .. Diez]`. Definir además el tipo `Cantidad` como sinónimo de `Int`.

b) Definir la función:

```
titulo :: Moneda -> String
```

que dada una moneda `m` devuelve una de las siguientes cadenas descriptivas según corresponda: `"Un peso"`, `"Dos pesos"`, `"Cinco pesos"`, `"Diez pesos"`.

No incluir el tipo `Moneda` en la clase `Eq`, ni en ninguna otra clase que no se haya pedido en el enunciado.

Ejercicio 2:

a) Definir el tipo `ContadoraMonedas` cuyos constructores son:
   - `Agregar`: Tiene dos parámetros. Guarda qué moneda se está agregando a la contadora (un elemento del tipo `Moneda`) y un valor del tipo `ContadoraMonedas` al cual se le añade el nuevo elemento.
   - `SinPlata`: No tiene parámetros y representa a una contadora que no tiene ninguna moneda.

b) Programar la función:

```
entregar_monedas :: ContadoraMonedas -> Moneda -> [Moneda]
```

que dada una contadora `cm` y una moneda `m` devuelve todas las monedas de la denominación de `m` dentro de la contadora `cm`.

Ejercicio 3:

a) Redefinir el tipo `ContadoraMonedas` como un sinónimo del tipo `ListaAsoc` que a cada `Moneda` le asocia una cantidad (algo de tipo `Cantidad`) que indica cuantas monedas de ese tipo se le agregaron.

b) Definir la función:

```
mas_de_cinco :: ContadoraMonedas -> [Moneda]
```

que dada una contadora `cm` devuelve una lista de monedas que tienen asociadas una cantidad mayor o igual a 5 unidades en `cm`.

Ejercicio 4*:

Definir la función

```
a_min :: Arbol a -> a
```

que dado un árbol `as` no vacío (`a_min Hoja` no debe estar definido) devuelve el elemento más chico dentro de `as`. Completar el tipado de la función para incluir los *type clases* necesarios para programarla.

Tema B:

Ejercicio 1:

a) Definir el tipo `Medalla` que consta de los constructores `Bronce`, `Plata` y `Oro`. Los constructores no toman parámetros. Al definir el tipo `Medalla` asegurarse que:
   - Esté en la clase `Show` (para que puedan mostrarse los valores del tipo)
   - Sus elementos **tengan definido un orden** (que la relación `<=` esté definida).

   Finalmente definir el tipo `Medallero` como un sinónimo de lista del tipo `Medalla` (es decir sinónimo de `[Medalla]`)

b) Definir la función:

```
valor_medalla :: Medalla -> Int
```

usando *pattern matching* que indica cuantos puntos vale una medalla:
   - Las medallas de bronce valen 1 punto
   - Las medallas de plata valen 2 puntos
   - Las medallas doradas valen 4 puntos

No incluir el tipo `Medalla` en la clase `Eq`, ni en ninguna otra clase que no se haya pedido en el enunciado.

Ejercicio 2:

a) Para llevar un registro de las medallas obtenidas en las distintas disciplinas durante un evento olímpico por nuestra delegación vamos a definir dos tipos:
   - El tipo `Disciplina` que tiene los constructores `Boxeo`, `Judo`, `Vela`, `Jockey` y `Tenis`.
   - El tipo `Resultado` que tiene el constructor `Res` que toma dos parámetros, el primero del tipo `Disciplina` y el segundo del tipo `Medalla`

b) Programar la función

```
medallero_vela :: [Resultado] -> Medallero
```

que dada una lista de resultados `xs`, devuelve las medallas obtenidas en la disciplina `Vela`.

No incluir el tipo `Disciplina` en la clase `Eq`, ni en ninguna otra clase que no se haya pedido en el enunciado.

Ejercicio 3:

a) Utilizando el tipo `ListaAsoc` definido en el ejercicio *6)* del *Proyecto 2* definir mediante recursión la función:

```
la_existe :: ListaAsoc a b -> a -> Bool
```

que dada una lista de asociaciones `la` y una clave `k` indica si dicha clave está en la lista `la`. Completar el tipado de la función incluyendo al tipo `a` en las clases que se necesiten para programarla.

b) Programar por composición, usando `la_busca` (definida en el *ejercicio 6b* del *Proyecto 2*) la función

```
la_existe' :: ListaAsoc a b -> a -> Bool
```

que al igual que `la_existe`, dada una lista de asociaciones indica si una clave se encuentra en ella.

Ejercicio 4*:

Programar la función

```
arbol_busca :: Arbol (Int, String) -> Int - > Maybe String
```

que dado un árbol `as` y un entero `k` devuelve la cadena que aparece asociada a `k` dentro del árbol `as`, o `Nothing` si no existe esa clave.

Tema C:

Ejercicio 1:

a) Definir el tipo `Verdura` cuyos constructores son: `Papa`, `Batata`, `Calabacin`, `Cebolla`. Asegurarse que el tipo esté en la clase `Show`. Definir además el tipo `Unidades` como sinónimo de `Int`.

b) Definir usando *pattern matching* la función:

```
titulo :: Verdura -> String
```

que dada una verdura `v` devuelve una de las siguientes cadenas descriptivas según corresponda: `"Papa Blanca"`, `"Batata Colorada"`, `"Calabacin coreanito"` y `"Cebolla morada"`

No incluir el tipo `Verdura` en la clase `Eq`, ni en ninguna otra clase que no se haya pedido en el enunciado.

Ejercicio 2:

a) Definir el tipo `Verduleria` cuyos constructores son:

- `Agregar`: Tiene dos parámetros. Guarda qué verdura se está agregando a la verdulería (un elemento del tipo `Verdura`) y un valor del tipo `Verduleria` al cual se le añade el nuevo elemento.
- `NoQuedaNada`: No tiene parámetros y representa a una verdulería sin *stock*.

b) Programar la función:

```
hay_verdura :: Verduleria -> Verdura -> Bool
```

que dada una verdulería `vs` y una verdura `v` devuelve `True` si y solo si hay en la verdulería *stock* de la verdura `v`.

Ejercicio 3:

a) Redefinir el tipo `Verduleria` como un sinónimo del tipo `ListaAsoc` que a cada `Verdura` se le asocia la cantidad (algo de tipo `Unidades`) con la que se cuenta.

b) Definir la función:

```
verduras_en_stock :: Verduleria -> [Verdura]
```

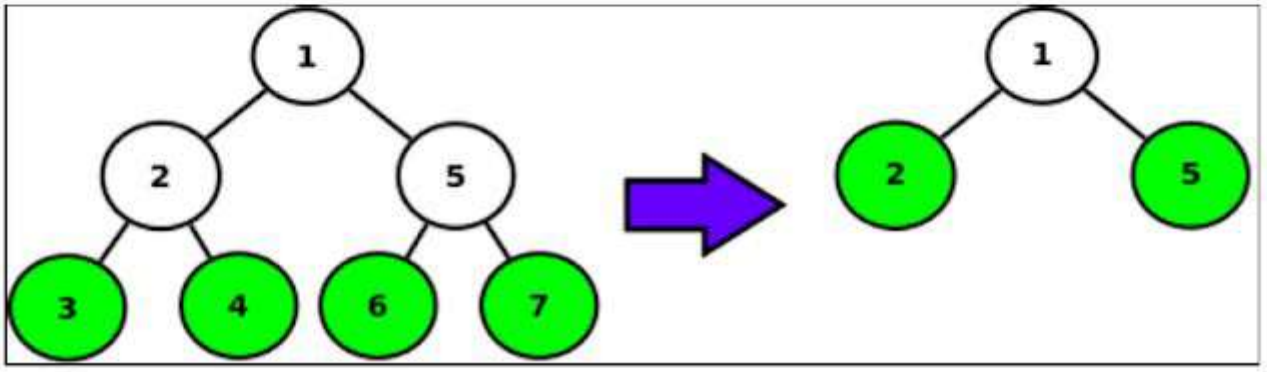
que dada una verdulería `vs` devuelve una lista de verduras que tienen un número disponible de unidades.

Ejercicio 4*:

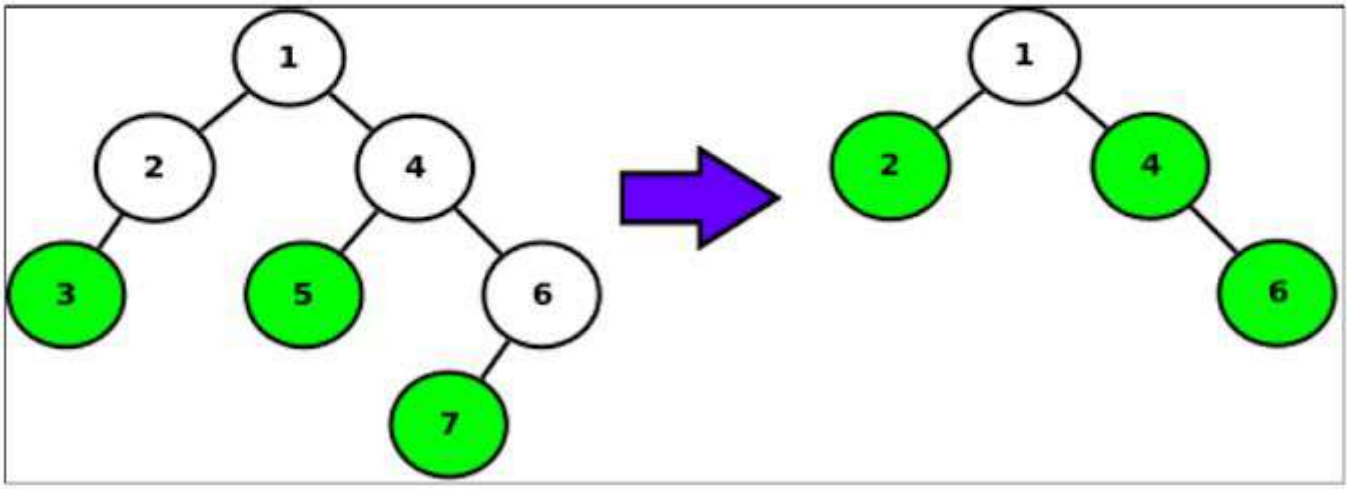
Definir la función

```
a_podar :: Arbol a -> Arbol a
```

que dado un árbol `as` no vacío (`a_podar Hoja` no debe estar definido) devuelve el un nuevo árbol cuyas ramas finales se han eliminado. Por ejemplo:



otro ejemplo:



Tema D:

Ejercicio 1:

a) Definir el tipo `NotaMusical` que consta de los constructores `Do`, `Re`, `Mi`, `Fa`, `Sol`, `La`, `Si`. Los constructores no toman parámetros. Al definir el tipo `NotaMusical` asegurarse que:
   - Esté en la clase `Show` (para que puedan mostrarse los valores del tipo)
   - Sus elementos **tengan definido un orden** (que la relación <= esté definida).

   Finalmente definir el tipo `Melodia` como un sinónimo de lista del tipo `NotaMusical` (es decir `[NotaMusical]`)

b) El sistema de notación musical anglosajón (conocido como notación o cifrado americano) relaciona las notas con letras de la A a la G. Programar mediante *pattern matching* la función:

```
cifrado_americano :: NotaMusical -> Char
```

que relaciona las notas do, re, mi, fa, sol, la, si con los caracteres `'C'`, `'D'`, `'E'`, `'F'`, `'G'`, `'A'`, `'B'` respectivamente.

No incluir el tipo `NotaMusical` en la clase `Eq`, ni en ninguna otra clase que no se haya pedido en el enunciado.

Ejercicio 2:

a) Ahora vamos a representar los acordes musicales (que se forman a partir de una nota). Hay varios tipos de acordes, entre ellos los acordes mayores y menores. Deben definir entonces los siguientes tipos:
   - El tipo `Modo` que tiene los constructores `Mayor` y `Menor` (que no toman parámetros)
   - El tipo `Acorde` que tiene un único constructor `Cons` que toma dos parámetros, el primero del tipo `NotaMusical` y el segundo del tipo `Modo`

b) Programar la función

```
solo_mayores :: [Acorde] -> [Acorde]
```

que dada una lista de acordes `as`, devuelve una lista los acordes de `as` que tienen modo `Mayor`.

No incluir el tipo `Modo` en la clase `Eq`, ni en ninguna otra clase que no se haya pedido en el enunciado.

Ejercicio 3:

a) Utilizando el tipo `Cola` definido en el ejercicio *5)* del *Proyecto 2* definir mediante recursión la función:

```
existe :: Cola -> Cargo -> Bool
```

que dada una cola `cs` y un cargo `c` indica si hay un docente con cargo `c` dentro de la cola `cs`.

b) Programar por composición, usando a `busca` (definida en el ejercicio *5a* del *Proyecto 2*) la función

```
existe' :: Cola -> Cargo -> Bool
```

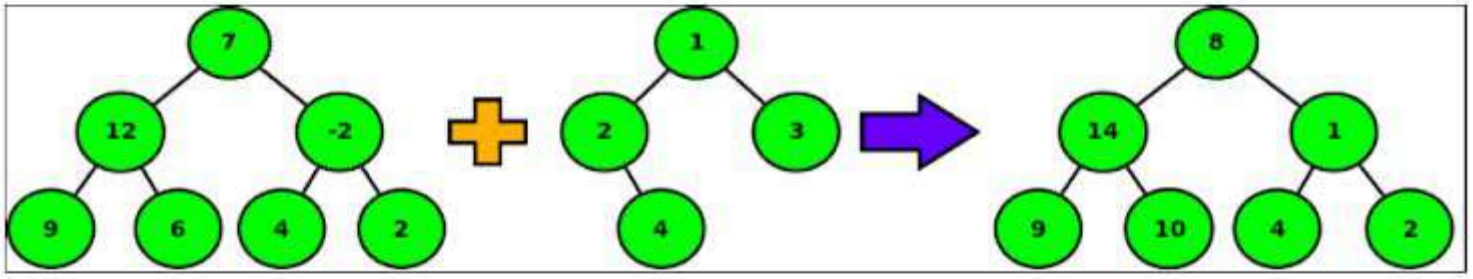
que indica lo mismo que la función original `existe`

Ejercicio 4*:

Programar la función

```
arbol_sum :: Arbol Int -> Arbol Int - > Arbol Int
```

que dado dos árboles `as` y `bs` devuelve un nuevo árbol cuyos valores son la suma de los elementos `as` y `bs` punto a punto. Ejemplo:



Además, la suma de árboles es conmutativa, o sea que: