Repaso de las clases anteriores

Rapidito… ¿qué vimos hasta ahora?

- Asignación destructiva / efecto colateral / transparencia referencial
- Declaratividad vs. Imperatividad
- Expresividad
- Funciones
- Tipos (+ definición de nuevos tipos)
- Chequeo de tipos de un lenguaje
  - estático
  - dinámico
- Composición
- Recursividad
- Listas / Listas por comprensión / Listas infinitas
- Tuplas
- Formas de evaluación:
  - ansiosa
  - perezosa → permite "listas infinitas"

Funciones de orden superior

Una función puede recibir como argumentos
1) números, booleanos, listas o…
2) ¡funciones!

Las primeras funciones se llaman funciones de primer orden.
Las funciones que reciben o devuelven funciones se llaman funciones de orden superior.

Darle a cada fila de bancos a que resuelvan sin usar listas por comprensión:
1) Dada una lista, devolver otra lista con los números mayores a n.

```
mayoresA n lista
```

2) Dada una lista, devolver otra lista con los números primos.

```
primos lista
```

3) Dada una lista, devolver otra lista con los números positivos.

```
positivos lista
```

Vemos que son todas iguales, es un embole → ¿y si abstraemos lo que se repite en el código?

Podríamos separar la misma función en dos objetivos diferentes:

1) filtrar los elementos de una lista (por un criterio).
2) indicar si un elemento cumple o no una determinada condición.

1) Filtrar elementos de una lista por un criterio. Ese criterio lo recibo como parámetro:

```
filter f [] = []
filter f (x:xs) | f x       = x : filter f xs
                | otherwise = filter f xs
```

Si usara listas por comprensión, ¿cómo sería?

```
filter f xs = [ x | x <- xs, f x ]
```

“El truco consiste en pasar la función como parámetro"
¿De qué tipo es filter?

Puedo recibir ¿cualquier función? No, una que reciba cualquier parámetro pero que devuelva un Bool.
Y una lista de cualquier parámetro, pero tiene que ser igual de las que pide la primera función.
¿Y qué devuelve? Una lista del mismo tipo que recibo.

O sea:
`filter :: (a -> Bool) -> [a] -> [a]`

El `(a -> Bool)` indica que recibo una función como parámetro.
En el repaso dijimos que cuando yo uso una función, en cada parámetro le paso una expresión.

- Las funciones son expresiones.

¿Cómo aplico los tres ejercicios anteriores?
Ejercicio 1) `mayoresA n xs = filter (> n) xs`
Ejercicio 2) `primos xs = filter primo xs`
Ejercicio 3) `positivos xs = filter (> 0) xs`

Estamos cerca del lenguaje coloquial: “`filter (> n) xs`: filtrame los que sean mayores a n de esta lista”.

En definitiva, estoy descomponiendo el problema en problemas más chicos.

Volvemos sobre expresiones
  - Se puede evaluar (y obtengo otra expresión, que puede ser un valor o… una función; veremos más sobre esto en breve)
  - Tiene un tipo
  - Es un valor (decirlo fuerte: ¡¡¡una función es un valor!!!)
  - 4 y min son valores. Lo único que no puedo hacer con el 4 que sí puedo hacer con min es pasarle parámetros.

- Esas cosas en las que min y 4 son iguales, es lo que le da power al paradigma.
- "Las funciones son valores de primer orden".

Composición y Orden Superior

¿Qué diferencia hay entre *`((>= 18).edad) persona`* y *`filter even xs`*, más allá de que tienen objetivos diferentes?

En la primera tenemos **composición de funciones**, tengo una expresión que se evalúa. El valor que le envío a `(>= 18)` es un entero, que es lo que se obtendrá de la reducción de la expresión `edad persona`.

En el segundo caso tenemos **una función como valor de primer orden**. O sea… le mando la función a `filter`, y `filter` adentro llama a esa función que le paso como parámetro. Es una función de orden superior (mientras que la primera no). Lo potente es que `filter` recibe una función que ni siquiera conoce. Simplemente la usa (delega la responsabilidad a la otra función).

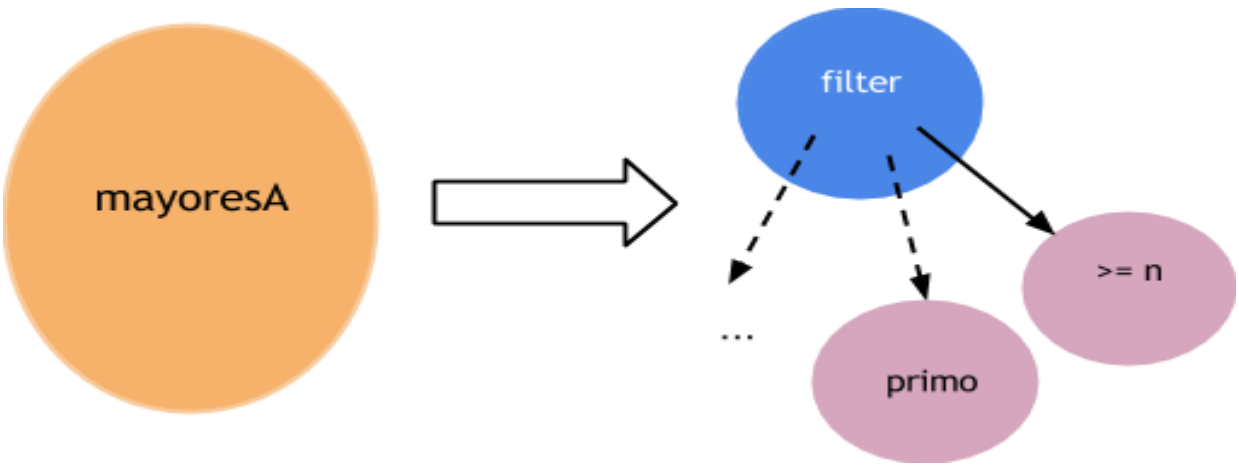
Ventaja de usar funciones de orden superior

Separo responsabilidades: si tuviera una consultora, podría decirle a 2 programadores distintos:

Esta es la base para dividir trabajo. ¿En qué necesitan ponerse de acuerdo los dos programadores?
En la definición de la función que determina el criterio, y más específicamente en su tipo:
`(a -> Bool)`
OK, listo, los dos ya pueden dedicarse a lo suyo.

`(a -> Bool)` implica que hay un **contrato** entre ambos programadores, es el punto de encuentro para coordinar ambas tareas.
Supongamos que el que hizo la función `primo` se equivocó y tiene que corregirla. ¿Hay que tocar `filter`? No, porque `filter` está **desacoplada** de la función `f` (conoce sólo el tipo de `f` y la usa, no le interesa cómo está implementada). Eso está bueno: evita tener que ir un fin de semana a arreglar código que dejó de andar.

Cuando hice `mayoresA`, positivos y primos tenía una función que hacía 2 cosas, ahora tengo dos funciones que son más genéricas y que hacen una sola cosa:



Eso facilita la prueba unitaria: puedo testear `primo` en forma independiente de `filter`. Si todo está en la misma función sólo puedo probar la función que filtra palíndromos en su totalidad.

Otra función de orden superior / map: transforma una lista en otra aplicando una función a todos sus elementos.

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

Usando listas por comprensión:
```
map f xs = [ f x | x <- xs ]
```

¿De qué tipo es `map`?
```
map :: (a -> b) -> [a] -> [b]
```

Entonces puedo multiplicar por dos los elementos de una lista.
```
> map (* 2) [1, 2, 3]
```

Y convertir una cadena de caracteres a mayúsculas:
```
upperCase palabra = map toUpper palabra
```

Ejemplos de map heterogéneo
Devolver la longitud (en caracteres) de una lista de palabras
```
sumarPalabras ["paradigmas", "rules", "the", "world"]
```

Pensemos cómo utilizar `map`:

1) necesitamos contar cuántas letras tiene cada palabra, mmmm… puede servirnos `map`. ¿Qué función nos devuelve la longitud de una cadena de caracteres?
2) necesitamos sumar las respectivas longitudes. ¿Qué función conocen que nos devuelva la sumatoria de una lista de enteros?

```
sumarPalabras palabras = sum (map length palabras) = (sum . map length) palabras
```

Pregunta:
¿qué devuelve?
```
map (+) [1, 2, 3]
```
Devuelve una lista de funciones:
```
[(1+), (2+), (3+)]
```

Para el lector: identifique dónde aparecen en la solución los conceptos

- composición
- función de orden superior

Nota: esto no lo van a poder ver así en Haskell, porque el motor no va a saber *mostrar* las funciones que componen la lista.

Ojo, yo puedo generar una función de orden superior sin necesidad de recibir una función:
```
f n = (+ n) : f (n + 1)
```

¿De qué tipo es?
```
f :: Int -> [Int -> Int]
```

Fíjense que lo que termino devolviendo es una lista de funciones.
Puedo invocarla haciendo:
`> (head . f 3)  2` ➔ me da 5.

¿Qué conceptos intervienen? Evaluación diferida que me permite tener listas infinitas (en la definición de `f`), funciones de orden superior (en la definición de `f`), composición (el resultado de `f 3` se compone con `head`) y… aplicación de funciones parciales (en breve conoceremos este concepto).

Bueno, piénsenlo y después la seguimos.

Apliquemos funciones de orden superior sobre otros tipos

Sobre tuplas

```
aplicarPar f (a,b) = (f a, f b)
```

Definamos una función para multiplicar por dos los elementos de una tupla:

```
doblePar = aplicarPar doble
triplePar = aplicarPar (3 *)
```

¿Qué es `(3 *)`? Una función que dado un número te devuelve ese número multiplicado por 3.

```
meterEnPar f (a,b) (c,d) = (f a c, f b d)
sumaPar = meterEnPar (+)
```

Sobre nuestras propias definiciones de tipos

Recordamos que los empleados pueden ser jefes o empleados comunes,y alguna definición sobre la manera de conocer su sueldo:

```
data Empleado = Comun Float String | Jefe Float Int String

sueldoDe (Comun sueldo _ )             = sueldo
sueldoDe (Jefe sueldo genteACargo _ ) = sueldo + genteACargo * 1000
```

Resolvemos el total de sueldos de los empleados sin necesidad de utilizar una función recursiva:

```
(sum . map sueldoDe) empleados
```

Ejercicio: Hacer una función que indique si todos los elementos de una lista cumplen una determinada condición:

```
> all even [1..3]
False
```

Hacer la función any, que indique si alguno de los elementos de una lista cumple una determinada condición.

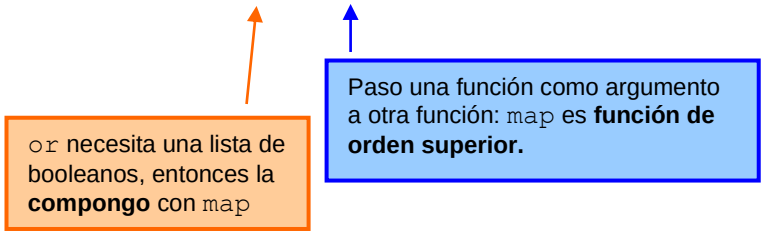
```
> any even [1..3]
True
```

Ayuda: hay una function `and :: [Bool] -> Bool`
que opera una lista de booleanos aplicándole un **`and`** entre sí.

```
any, all  :: (a -> Bool) -> [a] -> Bool
any f xs = or (map f xs)
all f xs = and (map f xs)
```

En definitiva, estamos usando composición y funciones de orden superior:

```
any f xs = (or . map f) xs
```



O bien…

```
any f = or . map f
```

(no escribimos `xs` porque lo infiere a partir de las definiciones de `map` y de `or`)

Otra opción:

```
all f xs = and [ f x | x <- xs ]
```

Y terminamos con `foldl`[1]:
¿Cómo era la suma de elementos de una lista?
¿Cómo era la multiplicación de los elementos de una lista?
¿Cómo concateno una lista de palabras?

(Darle 10' a que lo resuelvan)

```
sum  []     = 0
sum (x:xs) = x + sum xs

prod  []     = 1
prod (x:xs) = x * prod xs

concatenar []     = []
concatenar (x:xs) = x ++ concatenar xs
```

O bien…

```
sum  []     = 0
sum (x:xs) = (+) x (sum xs)

prod  []     = 1
prod (x:xs) = (*) x (prod xs)

concatenar []     = []
concatenar (x:xs) = (++) x (concatenar xs)
```

¡Ojo! No nos sirve dejar fijo el valor inicial. Tenemos que recibirlo como parámetro.

En términos generales… ¿cómo la paso a una función de orden superior?
Darles 5' a que piensen.

Necesitamos

- una función que opera con dos parámetros
- un valor inicial
- una lista

El resultado final sería la aplicación sucesiva de la función entre el valor inicial y cada uno de los elementos de esa lista.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Bueno, entonces, si lo recibimos como parámetro, ¿qué pasa en P(0)?

[1] La traducción más apropiada para el término fold es "reducir". Puede verse también http://www.haskell.org/haskellwiki/Fold y http://en.wikipedia.org/wiki/Fold_(higher-order_function)

Si la lista es vacía, devuelvo el “valor inicial”:

```
foldr f z []     = z
```

Si la lista no es vacía, ¿qué voy a tener que hacer?

```
foldr f z (x:xs) = f x (foldl f z xs)
```

Sumando con `foldr`:

```
sum = foldr (+) 0
```

Para sumar una lista hay que aplicar la suma elemento por elemento arrancando de 0.
Como es difícil de asimilar de golpe, vamos a hacer el seguimiento de un caso:

```
sum = foldr (+) 0

> sum [2, 3, 5]
= foldr (+) [2, 3, 5]
= (2 +) (foldr (+) 0 [3, 5])
= (2 +) ((3 +) (foldr (+) 0 [5]))
= (2 +) ((3 +) ((5 +) (foldr (+) 0 [])))
= (2 +) ((3 +) ((5 +) 0))  ← caso base, aquí uso el valor inicial
= (2 +) ((3 +) 5)
= (2 +) 8
= 10
```

Si revisan el Prelude, van a encontrar otra función similar:

```
foldl f z []      = z
foldl f z (x:xs)  = foldl f (f z x) xs
```

foldl trabaja asociando a izquierda la función f (mientras que foldr asocia a derecha, por eso **fold l**eft y **fold r**ight). El seguimiento puede ayudar a entender la diferencia:

```
> sum [2, 3, 5]
= foldl (+) 0 [2, 3, 5]
= foldl (+) ((+) 0 2) [3, 5]
= foldl (+) ((+) 2 3) [5]
= foldl (+) ((+) 5 5) []
= foldl (+) 10 []
= 10  ← caso base
```

Como se asocia a izquierda, z termina siendo el valor final, más que el inicial (vean el caso base)

¿Cuál es la diferencia? ¡Revisemos los tipos!

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Con las funciones `f` para las cuales el orden de sus parámetros no altera el resultado (como `(+)` y `(*)`), a efectos prácticos no hay diferencia. En las funciones que esto no sucede (como `(/)`), entonces se debe elegir la correcta.

Otros ejercicios que aplican `foldl`:

```
product = foldl (*) 1
product [1,4,5,6]
120

concatenar = foldl (++) []
```

Para concatenar una lista hay que aplicar el operador de concatenación `(++)` arrancando de una cadena vacía.