Repaso de las clases anteriores

Rapidito… ¿qué vimos hasta ahora?
Asociados a un paradigma

- Que tenga o no asignación destructiva / efecto colateral / transparencia referencial
- Que sea más o menos declarativo

Respecto a los lenguajes

- Chequeo de tipos
  - estático
  - dinámico

Respecto a la programación funcional en particular

- Funciones
- Tipos
- 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.

Si queremos resolver estos requerimientos sin 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 muy parecidas, es un embole → ¿y si abstraemos lo que se repite en el código?

```
quedarmeConlosQueCumplanUnCriterioX [] = []
quedarmeConQueCumplanUnCriterioX (x:xs)
    | cumpleCriterioX x = x : quedarmeConQueCumplanUnCriterioX xs
    | otherwise         = quedarmeConQueCumplanUnCriterioX xs
```

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

Qué cerca estamos 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".

Pregunta:

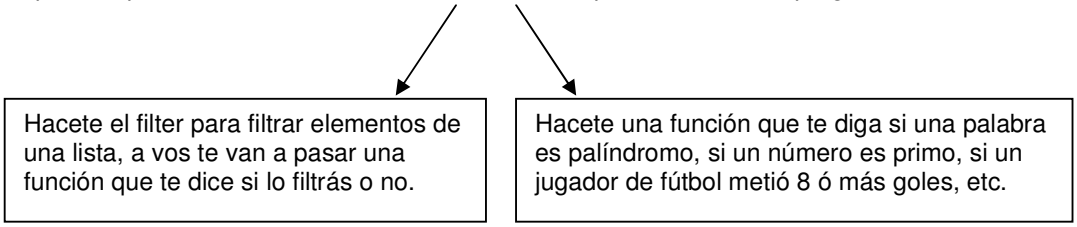
¿Qué diferencia hay entre `(even . enesimo n) xs` y `filter even xs`, más allá de que tienen objetivos diferentes?

En la primera tenemos composición de funciones, tengo una expresión que se evalúa. El valor que le envío a even es un entero, que es lo que se obtendrá de la reducción de la expresión enesimo n xs.

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

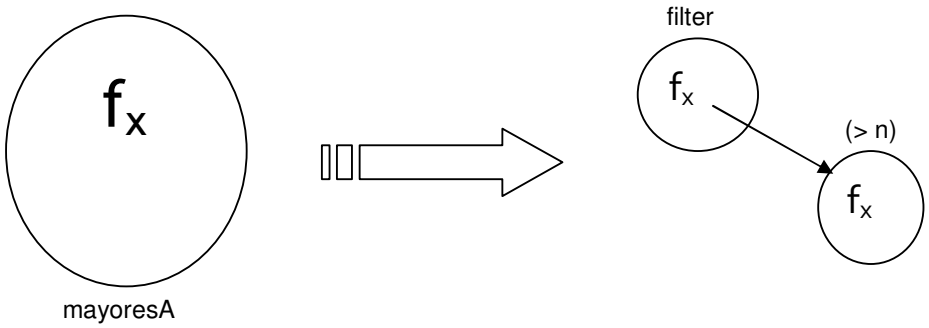
Separo responsabilidades: si tuviera una consultora, podría decirle a 2 programadores distintos:



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 palindromo 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.

Cuando hice mayoresA, positivos y primos tenía una función que hacía 2 cosas, ahora tengo dos funciones que hacen una sola cosa y que puedo utilizar en diferentes contextos:



Y además tener más funciones facilita la prueba unitaria: puedo testear `palindromo` en forma independiente de `filter`. Si todo está en la misma función sólo puedo probar la función que filtra palíndromos en su totalidad.

Otra fn 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
```

o utilizando eta reduction (notación pointfree):

```
upperCase palabra = map toUpper palabra
upperCase = map toUpper
```

Ejemplo de map heterogéneo

Devolver la longitud (en caracteres) de una lista de palabras

```
>sumarPalabras ["paradigmas", "rules", "the", "world"]
23
```

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
sumarPalabras = sum . map length
```

Pregunta:

¿qué devuelve?

```
>map (+) [1, 2, 3]
```

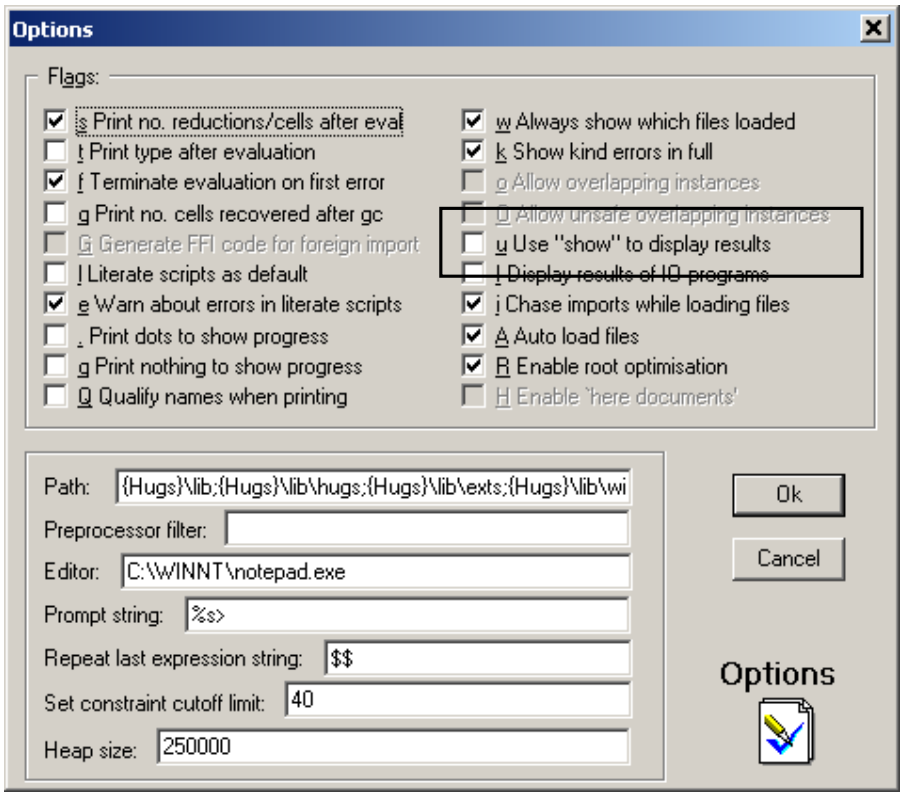
Devuelve una lista de funciones:

```
[ (+ 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, salvo que saquen el flag u para que no use Show al mostrarle los resultados.



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 función.
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 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 (+)
```

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 cumplen 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`
Lo resolvemos usando composición y funciones de orden superior:

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

Or necesita una lista de booleanos, entonces la **compongo** con map

Paso una función como argumento a otra función: map es **función de orden superior.**

o bien `any = or . map`

Otra opción:

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

Fold

Y terminamos con fold[1] que permite procesar una estructura de datos para construir un valor.
¿Cómo sumo los elementos de una lista?
¿Cómo multiplico los elementos de una lista?
¿Cómo concateno una lista de palabras?

```
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.
Darles 5' a que piensen.

[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)

En términos generales… ¿cómo la paso a una función de orden superior?
Necesitamos

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

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

Caso base: si la lista es vacía, devuelvo el valor inicial.

```
foldr f valorInicial []      = valorInicial
```

Si la lista no es vacía, ¿qué voy a tener que hacer?
A partir de alguna de las funciones podemos trasladar:

```
concatenar           (x:xs) = (++) x (concatenar xs)
foldr f valorInicial (x:xs) = f x (foldr f valorInicial 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 [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
= (+ 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
```

¿Cuál es la diferencia? Revisemos los tipos:

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

Foldr trabaja asociando a derecha la función f, mientras que foldl trabaja asociando a izquierda la función f.

El seguimiento de la sumatoria definida en base a foldl puede ayudar a entender cómo realiza la operatoria:

```
sum = foldl (+) 0

> 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
```

Y aquí vemos que el valor inicial del caso base es en realidad el resultado final.

Foldl vs. foldr

Con las funciones f para las cuales el orden de sus parámetros no altera el resultado, a efectos prácticos no hay diferencia. En las funciones que esto no sucede, entonces se debe elegir la correcta.

Otra diferencia entre fold a izquierda y derecha es el tratamiento con listas potencialmente infinitas. Recordemos las dos definiciones:

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

Consideremos las siguientes funciones:

```
elPrimo x y | primo x   = x
            | otherwise = y

primo x = -- una definición más cool,
          -- un número es primo si no hay antecesores divisibles
   null [antecesor | antecesor <- [2..x - 1], esDivisible x antecesor]

esDivisible x antecesor = mod x antecesor == 0
```

¿Qué sucede al querer evaluar
`foldr elPrimo 0 [8..]` vs. `foldl elPrimo 0 [8..]` ? ¿Por qué?

Otros ejercicios que aplican fold:

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

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

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