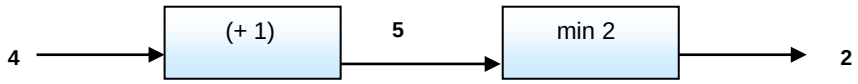
Repaso general de las clases anteriores

¿Qué vimos hasta ahora?... (Sí, siempre empezamos con esto ☺)

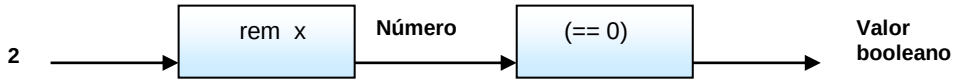
- Asignación destructiva / efecto colateral / transparencia referencial
- Declaratividad vs. Imperatividad
- Expresividad
- Funciones
- Tipos (+ definición)
- Chequeo de tipos de un lenguaje
  - estático
  - dinámico
- Composición
- Recursividad
- Listas / Listas por comprensión / Listas infinitas
-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

Repaso de composición

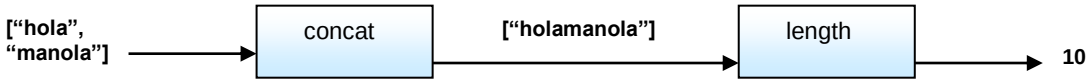
Cuando hago
(min 2 . (+ 1)) 4
¿Qué estoy haciendo?
Lo que me devuelve 4 + 1 lo compongo con min 2.



Si yo tengo
((== 0) . rem x) 2
¿Qué funciones compongo?



Para obtener la cantidad de letras de una frase:
(sum . map length) [“hola”, “manola”] o
(length. concat) [“hola”, “manola”].



Funciones constantes

Muchas veces necesitamos usar una lista de valores en varias funciones, y nos resulta un embole tener que repetir n veces la misma lista:
menor [1, 5, 4, 3, 2, 7]
mayor [1, 5, 4, 3, 2, 7]
head [1, 5, 4, 3, 2, 7]

tail [1, 5, 4, 3, 2, 7]
etc. etc.
Entonces podemos aplicar una función constante: f(x) = k. En realidad no tenemos argumentos, eso nos permite darle un nombre representativo a un juego de datos particular:

jugadores = [“riquelme”, “abbondanzieri”, “ibarra”]

Ni qué hablar si esa lista es una lista de tuplas:
Un jugador viene dado por una tupla que tiene: nombre, partidos jugados, goles convertidos, veces que fue expulsado y edad.
jugadores = [(“riquelme”, 10, 6, 1, 26), (“abbondanzieri”, 10, 0, 0, 33), (“ibarra”, 10, 1, 0, 34)]

Entonces podemos pedir:
cantidad jugadores (es el length)
goleador jugadores (es el que metió más goles de la lista de jugadores)
etc.
¡nos acercamos a castellano!
Por supuesto, esto vale tanto para listas como para cualquier otro valor.

Repaso foldl

¿Cómo resuelvo el menor de una lista con foldl?

```
foldl (una función) (valor inicial) (lista)
```

La función… y es el menor.
El valor inicial tiene que estar en la lista: entonces le voy a poner el primer elemento como menor tentativo.
Y después le paso la lista.

```
menor xs = foldl min (head xs) xs
```

También podría hacer:

```
menor (x:xs) = foldl min x xs
```

Otros usos de foldl:

```
factorial n = foldl (*) 1  [1..n]
sum n = foldl (+) 0 [1..n]
```

Definiciones locales / Expresiones lambda

Haskell está basado en cálculo lambda, que es un sistema de reglas de transformación o reductor de expresiones que diseñó Alonzo Church.

λ x : x * x ← es una expresión lambda que denota la función cuadrado

En Haskell se codifica así:

```
(\x -> x * x)
```

Y se ejecuta:

```
(\x -> x * x) 2
4
```

Otro ejemplo:

```
? (\x y -> x + y) 2 3
5
```

Esto me permite definir funciones “anónimas”, que no tienen nombre y se usan en un contexto limitado (el de la misma función que estoy definiendo).
¿Qué desventaja tiene? El objetivo está dentro de la función y tiene menos expresividad que tener una función doble definida. La ventaja es que no tengo que definir funciones auxiliares cuando sólo lo necesito usar una vez.

Definición local de una función

Otra forma de escribir lo mismo

```
sumar 2 3 where sumar x y = x + y
```

¿Qué tiene de piola?
Si quiero saber el número de raíces de una ecuación cuadrática:
$ax^2 + bx - c = 0$

```
numeroDeRaices a b c
      | discriminante > 0  = 2
      | discriminante == 0 = 1
      | discriminante < 0  = 0
  where discriminante = b*b - 4*a*c
```

Defino una expresión en un solo lugar. Tiene un nombre explícito, lo que ayuda a su comprensión y la ventaja de no tener que definirse por “afuera”, aunque también sirve solamente en el contexto local de una función.

Combinación: foldl + expresiones lambda/definición local

Poder definir funciones anónimas me salva de tener que definir una función para un objetivo solo.
Ejemplo: tengo empleados en una compañía. Cada empleado es una tupla que me dice: su nombre, cuánto gana y la cantidad de hijos a cargo que tiene. Quiero saber:

1) el monto total de sueldos que debo pagar a fin de mes.
2) Para el día del niño estoy organizando un evento, entonces necesito saber la cantidad de hijos a cargo que hay en la compañía.

Tenemos la lista de empleados en esta definición:
empleados = [(“perez”, 1600, 2), (“lajmanovich”, 980, 5), (“drot”, 2000, 3), (“zeman”, 600, 0)]

Si lo quiero hacer con foldl, ¡no puedo usar el operador (+), porque el (+) suma números!
Pero revisando la definición de foldl:

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

Fijense que va de a en b en a. Y la lista que recibo es de bs, entonces el primer argumento es donde puedo ir sumando los sueldos (porque voy a devolver el acumulado de sueldos, que es de tipo a).

Una opción:

```
montoTotalSueldos empleados =
     foldl (\acum empleado -> acum + sueldo empleado) 0 empleados
```

…y defino sueldo como el snd, pero para una tupla de 3 elementos:

```
sueldo (_, monto, _) = monto
```

Acá sí terminamos generando otra función, pero es más probable que `sueldo` se termine usando más de una vez.

Otra opción (adaptar mediante *pattern matching*):

```
montoTotalSueldos empleados =
       foldl (\acum (_,sueldo,_) -> acum + sueldo) 0 empleados
```

Y pido:

```
>montoTotalSueldos empleados
5180
```

La segunda opción es fácil una vez que me acostumbré, ahora lo usamos con una definición local:

```
cantidadHijos empleados = foldl sumarHijos 0 empleados
     where sumarHijos acum (_, _, cantHijos) = acum + cantHijos
```

Lo usamos:

```
>cantidadHijos empleados
10
```

Más sobre pattern matching, expresiones lambda

El ejemplo anterior me muestra que a veces el pattern matching me salva de tener que definir una función que voy a usar sólo una vez. Definamos ahora la función `viejos`, que dada una dupla con el nombre de una persona y su edad, me devuelve los nombres de las personas que tienen más de 30 años.

```
> viejos [("dodine", 33), ("leo", 23), ("flor", 21)]

viejos xs = [ nombre | (nombre, edad) <- xs, edad > 30 ]
```

También puedo usar una expresión lambda:

```
viejos xs = filter (\(_, edad) -> edad > 30) xs
```

¡Momento!... Falta obtener el nombre. También puedo hacerlo con lambda:

```
viejos xs = (map (\(nombre, _) -> nombre) . filter (\(_, edad) -> edad > 30)) xs
```

En la primera opción usé listas por comprensión, en la segunda funciones de orden superior.

Otras formas de definir funciones: aplicación parcial

doble = (2 *), no necesitamos agregar el n. Asume:
doble n = (*) 2 n

¿Por qué?
(2 *) es una función que dado un número lo multiplica por 2. Esta es la magia que permite aplicaciones parciales de una función:
¿De qué tipo es (*)?
Numero -> Numero -> Numero

¿Cómo lo ven?
(Numero -> Numero) -> Numero
o
Numero -> (Numero -> Numero)

Bueno, la realidad es que es: Numero -> (Numero -> Numero).

¿Cómo se entiende esto? Asocio a derecha, así puedo armar una función aplicando parcialmente sus argumentos:

Si el tipo de (*) es Numero -> (Numero -> Numero)
El tipo de (2 *) es Numero -> Numero, o sea (* 2) me devuelve una función (que dado un número te devuelve el doble de ese número).

Lo mismo pasa con (1 +). Me devuelve una función, que dado un número te devuelve ese número más 1.
masUno = (1 +)
¿Si lo escribo como expresión lambda?
(\n -> 1 + n)

¿Se va entendiendo?
Recordemos entonces:

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

Devuelve una lista de funciones:

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

Puedo componerlo con head:

```
(head . map (+)) [1, 2, 3]
```

¿Qué me devuelve? La función (1 +)
¿Y si le pido

```
((head.map (+)) [1, 2, 3]) 7
```

?
8

¿Qué conceptos entraron en juego en este ejemplo?

- Funciones de orden superior (map)
- Composición: head compuesto con map (+)
- Mmm… ¿y cuando me queda (1 +) después de evaluar el head de la lista de funciones? Eso me devuelve una función parcialmente aplicada[1].

Vamos con otro ejemplo, de a poquito:

```
mayor :: Ord a => a -> a -> a
mayor a b | a > b     = a
          | otherwise = b
```

¿Qué me devuelve mayor 8? Me devuelve una función a la que le puedo pasar un número. Entonces devuelve el mayor entre ese número y 8.

```
mayor :: Ord a => a -> a -> a
mayor 8 :: (Ord a, Num a) => a -> a
mayor 8 4 :: (Ord a, Num a) => a
```

Dependiendo de la cantidad de parámetros obtengo una función o un valor resultante de la expresión calculada. Como las funciones son valores de primer orden (recordar el módulo 5) la magia del paradigma hace que no me de cuenta, y que pueda pasar funciones como parámetro sin problemas.

Dicho de otra manera: la función (mayor 8) queda evaluada parcialmente, sólo cuando le agrego el parámetro 4 se transforma en un valor numérico. Antes era también un valor de primer orden, pero *una función*. Esto no lo puedo hacer en C, y es otra diferencia fundamental entre:
mayor(8, 4) de C y

[1] Si alguien dice evaluación diferida, no está mal, siempre entra en juego; nos fijamos si ese concepto juega un rol importante en la solución: en este caso no así que lo pasamos por alto

mayor 8 4 de Haskell. Yo puedo dejar “por la mitad” mayor 8, mientras que eso no puede suceder en C.

Este concepto se llama aplicación parcial.

¿Qué ventaja tiene dejar una función parcialmente aplicada? **Puedo generar nuevas funciones**.

Definimos una función en base a otra:
```
cuadrado = (^ 2)
cubo = (^ 3)
```

Nota: Solo para el caso de los operadores como pueden ser (+), (*), (^), también vale pasar el segundo argumento como se muestra en estos ejemplos. Tener en cuenta que en los ejemplos anteriores ((+) y (*)) los parámetros de los operadores son conmutables (a + b = b + a) y por lo tanto (2+) y (+2) es lo mismo. La potenciación (^) no lo es, y por lo tanto es diferente (2^) que (^2).

Aplicación parcial (con el título puesto)

Quiero definir una función que me diga si hay algún múltiplo de un número en una lista.

```
>algunoEsMultiploDe 8 [3, 4, 5, 6]
True
```

Tengo una función que me dice si un número es múltiplo de otro:
```
multiploDe multiplo numero = mod multiplo numero == 0
```

multiploDe 8 : me dice si un número es múltiplo de 8.
Recibo un número y me devuelve un booleano.

Podemos usar la función any (de orden superior):
```
algunoEsMultiploDe multiplo numeros = any (multiploDe multiplo) numeros
```

Tengo que pasarle una función que necesita un solo argumento pero multiploDe tiene dos argumentos. Necesito una función con uno solo, el primero (x) es siempre el mismo.
Se puede hacer con una lambda o con una definición local.
Pero vemos que esto es más piola. Entonces, la aplicación parcial de funciones me permite generar nuevas funciones muy fácilmente, simplemente armo otra con un parámetro menos.

Otro ejemplo: buscar los divisores de 15 usando filter.

Una opción:
```
divisores n = divisoresDe n [1..n]
divisoresDe numero numeros = [ x | x <- numeros, mod numero x == 0 ]
```

Lo usamos:
```
Main> divisoresDe 15
[1,3,5,15]
```

¿Qué otra tengo?
```
divisoresDe divisor numeros = filter (divisorDe divisor) numeros
```

Invoco a una función que es `divisorDe x`, donde x está fijo, lo que voy variando son los elementos de la lista:
```
divisorDe divisor numero = mod numero divisor == 0
```

Y la verdad es que no querría definir una función `divisorDe` que hace lo mismo que `multiploDe` pero con los argumentos cambiados.
Hay justamente una función que te invierte los argumentos y evalúa una expresión que le pases como parámetro: se llama `flip`. ¿Cómo sería?

```
flip f x y = f y x
```

¿Qué tipos recibimos?

```
flip :: (a -> b -> c) -> b -> a -> c
```

Entonces podríamos hacer: `divisorDe x y = flip multiploDe x y`

O bien:

```
divisorDe ~~x y~~ = flip multiploDe ~~x y~~
```

Quedando:

```
divisorDe = flip multiploDe
```

`divisorDe 2 4` me da `True` (2 es divisor de 4).

Y usamos `divisoresDe`:

```
Main> divisoresDe 2 [1, 4, 5, 8, 9]
[4, 8]
```

Recordar: Volver sobre el filter de los mayores a 4.
`mayoresA n xs = filter (> n) xs` → estuvimos usando funciones parcialmente aplicadas