Paradigma Funcional - Clase 5

"Algoritmos genéricos en funciones de orden superior"

```
-- Vamos a representar un plato por una tupla de 3 elementos
(NombrePlato,PrecioPlato,[Ingrediente])
-- Vamos a representar a un ingrediente por una tupla de 2 elementos
(NombreIngrediente,CantidadGramos)

recomendacionesDelChef = [
    (("tallarines con mariscos",45, [("tallarines", 200),("salsa de tomate",
250), ("mejillones", 100), ("langostinos", 50), ("calamar", 100)]),
    (("risotto del bosque",32, [("arroz", 250), ("champignones", 100),
("hongos de pino",100), ("azafran",5)]),
    (("pato a la naranja",666, [("pato", 1000), ("naranja", 250)]),
  ...
]

-- cantidadCalorias cantidadGramos nombreIngrediente
cantidadCalorias "tallarines" = 287
cantidadCalorias "salsa de tomate" = 27
cantidadCalorias "mejillones" = 95
cantidadCalorias "langostinos" = 98
cantidadCalorias "calamar" = 78

cantidadCalorias "arroz" = 82
cantidadCalorias "champignones" = 40
cantidadCalorias "hongos de pino" = 0
cantidadCalorias "azafran" = 368

cantidadCalorias "pato" = 200
cantidadCalorias "naranja" = 42

ingredientes (_,_,ingredientesPlato) = ingredientesPlato
precio (_,platita,_) = platita

calorias (nombreIngrediente,gramos) = div (gramos * cantidadCalorias
nombreIngrediente) 100
```

Queremos saber cuál plato nos conviene según distintos caprichos:

- El plato más caro:

```
platoMasCaro :: [Plato] -> Plato
platoMasCaro recomendaciones =
    find (\elPlatoMasCaro -> all (\plato -> precio elPlatoMasCaro
>= precio plato ) recomendaciones ) recomendaciones
```

- El plato con más ingredientes - porque nos gusta la variedad (?):

```
platoConMasIngredientes :: [Plato] -> Plato
```

```
platoConMasIngredientes recomendaciones =
    find (\elPlatoConMasIngredientes -> all (\plato ->
(length.ingredientes) elPlatoConMasIngredientes
>= (length.ingredientes) plato ) recomendaciones ) recomendaciones
```

... Y para los que están haciendo dieta y quieren alejarse de ESE plato peligroso:

- El plato con más calorías

```
platoMasBomba :: [Plato] -> Plato
platoMasBomba recomendaciones =
    find (\elPlatoMasBomba -> all (\plato -> (sum.(map
calorias).ingredientes) elPlatoMasBomba >= (sum.(map
calorias).ingredientes) plato) recomendaciones ) recomendaciones
```

Si prestamos atención a estas 3 funciones, podemos ver que tienen la misma forma.

Las 3 están buscando al plato que "maximiza una función".

Entonces podríamos buscar una abstracción para una función que me busque *cuál plato de una lista es el mejor* (el que aplicado a la función dé el valor máximo):

```
platoMasIdealSegun :: [a] -> a
platoMasIdealSegun funcion recomendaciones =
    find (\elPlatoMasIdeal -> all (\plato -> funcion elPlatoMasIdeal
>= funcion plato) recomendaciones ) recomendaciones
```

De esa manera logramos tener una sola función para buscar el plato "máximo" según una función enviada como parámetro.

~~Nota~~: Vean que el dominio de `platoMasIdealSegun` es distinto del dominio de las funciones `platoMasBarato`, `platoConMasIngredientes` y `platoMasBomba`.

`[a] -> a` vs `[Plato] -> Plato`

Cómo **`funcion`** llega por parámetro, en esta definición no se habla específicamente de su dominio, solo sabemos que recibe como parámetro un elemento de la lista `recomendaciones`.

Nos queda entonces nuestra función genérica para buscar el elemento de una lista que hace máxima a una función:

```
maximoSegun funcion lista =
    find (\maximoElemento -> all (\elemento -> funcion elemento <= funcion
maximoElemento ) lista ) lista
```

Atención! Como comparamos 2 elementos para ver cuál es mayor, la imagen de **`funcion`** está restringida a valores **`Ord`**enables.

¿Cuál es el tipo de maximoSegun?

```
maximoSegun :: Ord a => (b -> a) -> [b] -> b
```

Entonces, usando esta función genérica `maximoSegun`, cómo sé cuál es el plato más barato?

```
> maximoSegun precio recomendacionesDelChef
("pato a la naranja",666, [("pato", 1000), ("naranja", 250)])
```

Y el plato con más ingredientes?

```
> maximoSegun (length.ingredientes) recomendacionesDelChef
("tallarines con mariscos",45, [("tallarines", 200),("salsa de tomate",
250), ("mejillones", 100), ("langostinos", 50), ("calamar", 100)])
```

Y el plato más bomba?

```
> maximoSegun (sum.(map calorias).ingredientes) recomendacionesDelChef
("pato a la naranja",666, [("pato", 1000), ("naranja", 250)])
```

¿Me sirve la función `maximoSegun` para saber cuál es el plato más barato? Un truquito:

El elemento mínimo de esta lista:
[1,3,5,8,10]

Es el valor absoluto del elemento máximo de esta otra lista:
[-1,-3,-5,-8,-10]

Pensando eso, podemos hacer la siguiente consulta para saber cuál es el plato más barato:

```
> maximoSegun ((-1*).precio) recomendacionesDelChef
("risotto del bosque",32, [("arroz", 250), ("champignones", 100), ("hongos
de pino",100), ("azafran",5)]),
```

Y también podemos preguntar cuál es el plato más light (esto es, el que tiene menos calorías):

```
> maximoSegun ((-1*).sum.(map calorias).ingredientes) recomendacionesDelChef
("risotto del bosque",32, [("arroz", 250), ("champignones", 100), ("hongos
de pino",100), ("azafran",5)]),
```

Otros ejemplos

A veces nos interesa saber si contamos con elementos que cumplen determinadas condiciones

```
caloriasPlato = sum.(map calorias).ingredientes

-- Ejemplo 1
hayEnElMenuUnPlatoConMasDe300Calorias recomendaciones =
    any ((300<).caloriasPlato) recomendaciones

-- Ejemplo 2
hayEnElMenuUnPlatoConMasDe2Ingredientes recomendaciones =
    any ((2<).length.ingredientes) recomendaciones

-- Ejemplo 3
hayEnElMenuUnPlatoConExactamente4IngredientesOQueContengaAzafran
recomendaciones =
```

```
    any (\plato -> ((4==).length.ingredientes) plato || (elem
"azafran".ingredientes) plato) recomendaciones
```

Podemos aplicar la misma idea que antes

```
cumplenAlgunaCondicion criterios unaLista = any (\f -> any f unaLista)
criterios

-- Ejemplo 1
> cumplenAlgunaCondicion [(300<).caloriasPlato] recomendaciones

-- Ejemplo 2
> cumplenAlgunaCondicion [(2<).length.ingredientes] recomendaciones

-- Ejemplo 3
> cumplenAlgunaCondicion [(4==).length.ingredientes, elem
"azafran".ingredientes ] recomendaciones
```

Recursividad

Recursividad:
  Vea Recursividad

Intentemos definir la función factorial.

$$n! = 1 \times 2 \times 3 \times 4 \times ... \times (n-1) \times n$$

$$n! = \begin{cases} \text{si } n = 0 & \Rightarrow 1 \\ \text{si } n \geqslant 1 & \Rightarrow (n-1)! \cdot n \end{cases}$$

Escribamos esto en un programa de Haskell:

```
factorial n = n * factorial (n-1)
```

¿Y qué pasa con el factorial de 0? Podemos resolverlo con pattern matching:

```
factorial 0 = 1
```

Podemos ver que la función `factorial` "se aplica a sí misma" para resolverse. Dicho de otra forma, la función `factorial` está definida en términos de sí misma.

Pensemos en cómo Haskell podría tratar de resolver una consulta sobre esta función:

```
> factorial 5

factorial 5 = 5 * factorial 4
factorial 5 = 5 * (4 * factorial 3)
factorial 5 = 5 * (4 * (3 * factorial 2))
factorial 5 = 5 * (4 * (3 * factorial 2))
factorial 5 = 5 * (4 * (3 * (2 * factorial 1)))
```

```
factorial 5 = 5 * (4 * (3 * (2 * (1 * factorial 0))))
factorial 5 = 5 * (4 * (3 * (2 * (1 * 1))))
...
factorial 5 = 120
```

También podríamos definir el producto en términos de sí mismo:

```
producto 0 _ = 0
producto m n = n + producto (m-1) n
```

Haskell lo podría resolver de la siguiente manera:

```
> producto 3 2

producto 3 2 = 2 +  producto 2 2
producto 3 2 = 2 + 2 + producto 1 2
producto 3 2 = 2 + 2 + 2 producto 0 2
producto 3 2 = 2 + 2 + 2 + 0
producto 3 2 = 6
```

La potencia también podría definirse en términos de sí misma:

```
potencia base 0 = 1
potencia base exp = base * potencia base (exp-1)
```

Viendo los ejemplos anteriores podemos ver que las funciones recursivas se define con al menos un término recursivo (**Caso Recursivo**), el cual incluye una llamada a sí misma y un término no recursivo (**Caso Base**) que se establece como condición de corte para que la función no entre en un ciclo infinito.

```
--CASO BASE
factorial 0 = 1
--CASO RECURSIVO
factorial n = n * factorial (n-1)

--CASO BASE
producto 0 _ = 0
--CASO RECURSIVO
producto m n = n + producto (m-1) n

--CASO BASE
potencia base 0 = 1
--CASO RECURSIVO
potencia base exp = base * potencia base (exp-1)
```

Con todo lo visto anteriormente podemos decir que usamos **recursividad** cuando definimos algo en términos de sí mismo.

Estructuras recursivas

Así como tenemos definiciones de funciones recursivas, podemos tener valores recursivos ... CHAN!

La **matrioska** o **muñeca rusa** (ruso: Матрёшка /m??tr?o?k?/) son unas muñecas tradicionales rusas creadas en 1890, cuya originalidad consiste en que se encuentran huecas por dentro, de tal manera que en su interior albergan una nueva muñeca, y ésta a su vez a otra, y ésta a su vez otra [...] ***Wikipedia***

Podemos ver que una matrioska está compuesta por muchas muñecas pero para entender realmente qué es una matrioska tenemos que contestar una simple pregunta ...

Existen 2 posibles respuestas

- Dentro de una muñeca hay otra muñeca o,
- Dentro de una muñeca no hay nada

Esto que parece trivial se cumple para todas las muñecas (la más grande, las del medio, la más chica).

Pero es importante entender que dentro de la muñeca más chica **no hay nada** y dentro de todas las otras muñecas **hay una muñeca**.

Si la flasheamos un rato y decimos "siempre dentro de una muñeca vamos a tener otra muñeca" ... cuántas muñecas forman mi matrioska? Claro, la ~~caguć~~ embarré. Tendría una cantidad infinita de muñecas!

En el ejemplo de la matrioska

Caso base: una muñeca tiene adentro nada
Caso recursivo: una muñeca tiene adentro otra muñeca

Si toda la definición es recursiva en si misma estamos involucrando el concepto de infinito, porque nunca se terminaría la definición. Por eso es necesario incluir dentro de la **definición recursiva** o **caso recursivo** algo NO recursivo el **caso base**.

Podemos pensar a **la lista** como una estructura definida recursivamente

Caso base: una lista sin ningún elemento (conjunto vacío)
Caso recursivo: una lista con un elemento y que le sigue otra lista

A la lista que no tiene elementos la vamos a llamar **lista vacía** y la representamos así **[]**

A las listas que tienen al menos un elemento las representamos como cabeza y cola así **(Cabeza:Cola)** donde **la Cola es otra lista** que debe cumplir con alguno de estos dos patrones.

Una lista ["ger","lider","leo"] se puede representar de las siguientes maneras

```
[ "ger" , "lider" , "leo" ]
 "ger" : [ "lider" , "leo" ]
 "ger" : ( "lider" : [ "leo" ] )
"ger" : ( "lider" : ( "leo" : [ ] ) )
```

Nota: Los paréntesis son a modo de mejorar el entendimiento, no son necesarios.

Por lo tanto, si son lo mismo, podríamos realizar la siguiente consulta:

```
> [ "ger" , "lider" , "leo" ] == ("ger" : "lider" : "leo" : [])
True
```

Efectivamente las dos expresiones representan exactamente el mismo valor.

Retomando maximoSegun

```
--Recibe una función (criterio) y una lista y devuelve el máximo de esa
lista según el criterio

maximoSegun :: Ord a => (b -> a) -> [b] -> b
maximoSegun _ [x] = x
maximoSegun f (x:xs)
    | f x > f (maximoSegun f xs) = x
    | otherwise = maximoSegun f xs
```

Que escrito de otra manera podría ser:

```
maximoSegun _ [x] = x
maximoSegun f (x:xs)
    | f x > f maximoResto = x
    | otherwise = maximoResto
    where maximoResto = maximoSegun f xs
```

En Haskell viene una función `id` que parece bastante inservible

```
id x = x
```

Pero, para conocer el número más grande de una lista podemos hacer:

```
> maximoSegun id [1,4,6,10,2,6]
10
```

El menor número de una lista

```
> maximoSegun ((-1)*) [1,4,6,10,2,6]
1
```

El alumno con legajo más chico

```
> maximoSegun ((*(-1)).snd)
[("adri",1229085),("german",1199234),("leo",1339075)]
("german",1199234)
```

Algunos ejemplos de recursividad con listas

```
--Recibe una lista y me devuelve un Int que representa la cantidad de
elementos que tiene
length [] = 0
length (_:xs) = 1 + length xs


--Recibe una lista de números y devuelve la suma de la misma
sum [] = 0
sum (x:xs) = x +  sum xs

--Recibe un elemento y una lista y dice si dicho elemento se encuentra en
la lista
elem _ [] = False
elem x (y:ys)
    | x == y = True
    | otherwise = elem x ys

--Recibe una lista de listas y me devuelve una lista que tiene todos los
elementos
concat [] = []
concat (xs:xss) = xs ++ (concat xss)

--Recibe una función (condición) y una lista y me devuelve True si la
condición se cumple para todos los elementos
all _ [] = True
all f (x:xs) = (f x) && (all f xs)

--Recibe una función (condición) y una lista y me devuelve True si la
condición se cumple para alguno de los elementos
any _ [] = False
any f (x:xs) = (f x) || (any f xs)


--Recibe una función (condición) y una lista y me devuelve una nueva lista
con los elementos de la primer lista que cumplen dicha condición
filter _ [] = []
filter f (x:xs)
    | f x = x : filter f xs
    | otherwise = filter f xs

--Recibe una función (condición) y una lista y me devuelve una nueva lista
con los elementos de la primer lista que cumplen dicha condición
map _ [] = []
map f (x:xs) = f x : map f xs

--Recibe una lista de Personas y me devuelve la suma de sus edades
sumatoriaEdades [] = 0
sumatoriaEdades (x:xs) = edad x + sumatoriaEdades xs
```

Fold

Algunas de las funciones definidas anteriormente se pueden ver de la siguiente manera:

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

concat [] = []
concat (xs:xss) = (++) xs (concat xss)


sumatoriaEdades [] = 0
sumatoriaEdades (x:xs) = (\total pers -> total + edad pers)
(sumatoriaEdades xs) x


--
```

Si miramos detenidamente nos daremos cuenta que todo lo que escribimos se parece a

```
funcionSaraza [] = valorInicial
funcionSaraza (elemento:restoElementos) =
      funcion elemento (funcionSaraza restoElementos)
```

Es importante ver **`valorInicial`** y **`funcion`** en los ejemplos

```
-- sum
valorInicial = 0
funcion = (+)

-- concat
valorInicial = []
funcion = (++)

-- sumatoriaEdades
valorInicial = 0
funcion = (\total pers -> total + edad pers)
```

Con todo esto podemos repensar las definiciones de `sum`, `concat` y `sumatoriaEdades`

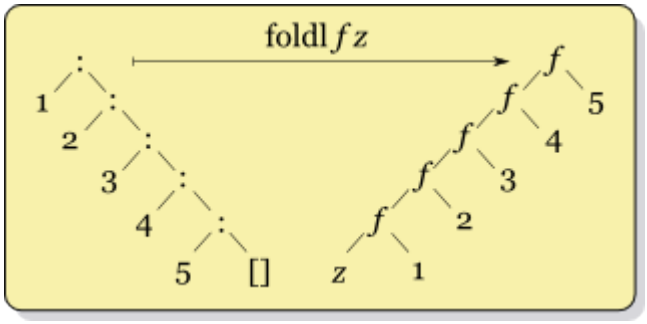
```
sum numeros = foldl (+) 0 numeros
concat listas = foldl (++) [] listas
sumatoriaEdades listaPersonas = foldl (\total pers -> total + edad pers) 0
listaPersonas

Otros Ejemplos:

all f lista = foldl (\acu el -> acu && f el) True lista
any f lista = foldl (\acu el -> acu || f el) False lista
```

Resumiendo:

foldl recibe
- una función de dos parámetros
  - el 1er parámetro es el valor actual del acumulador
  - el 2do parámetro es cada elemento de la lista
  - Retorna el nuevo valor del acumulador
- valor inicial del acumulador
- lista

foldl retorna el valor final del acumulador

La idea de foldl se puede ver como que "plega" (foldea) a izquierda (fold = foldl left)



Ejemplo

```
foldl (+) 0 [1,2,3,4,5]
```

Si tomamos a
- z como el valor inicial (en nuestro ejemplo `0`)
- f la función que le mandamos al foldl (en nuestro caso `+`)

```
[1,2,3,4,5] == 1:2:3:4:5:[]
```

-- Se comienza desde el elemento que está más a la izquierda (en nuestro caso 1)
`(((((0 + 1) + 2) + 3) + 4) + 5)`

Dándonos como resultado **15**

Otros ejemplos

```
maximo numeros@(x:_) = foldl max x numeros

-- Siendo curso :: [(Nombre,Legajo,Notas)]
promedioNotas curso =
  div (foldl (\total (_,_,notas)-> sum notas + total) 0 curso) (length
curso)
```

La versión anterior usa la división entera, si queremos la división "con coma" hacemos

```
promedioNotas curso =
  (foldl (\total (_,_,notas)-> sum notas + total) 0 curso) /
      fromIntegral (length curso)
```