Paradigma Funcional - Clase 3

Comprendiendo las listas

En matemática es común describir un conjunto de la siguiente forma:

```
{ x / x ∈ otroConjunto ... }
```

Ejemplo:

Si quiero decir que el conjunto A es igual a un conjunto con todos los números pares de un conjunto B, escribo lo siguiente:

```
A = { x / x ∈ B ∧ x = 2k con k ∈ ℤ}
```

O si nos alejamos un poco de la matemática alcanzaría con decir

```
A = { x / x ∈ B ∧ que x sea par }
```

Si quiero decir que el conjunto B es igual a un conjunto con todos los números de A pero cada uno de esos números debe estar multiplicado por 2 y sumado a 1 , escribo lo siguiente:

```
C = { 2x + 1 / x ∈ A }
```

O de forma similar:

```
C = { 2x + 1 / x ∈ B ∧ que x sea par }
```

Volvamos al ejemplo de los alumnos y los cursos (pueden ver el resumen de la clase 2 para refrescarlo)

- Qué pasa si queremos conocer los nombres de los alumnos de un curso que estén aprobados?

```
nombreAlumno unAlumno = fst unAlumno

notasAlumno unAlumno = snd unAlumno

estaAprobado unAlumno = all (>=4)(notasAlumno unAlumno)

--o bien
estaAprobado = (all (>=4)).notasAlumnos
```

Explicación

Componemos una función que recibe una lista y nos dice si todos los elementos son mayores a 4
`todosMayoresOIgualesQueCuatro lista = all (>=4) lista`
Con una función que recibe un alumno y nos devuelve su lista de notas
`notasAlumno unAlumno = snd unAlumno`

Si f es `todosMayoresACuatro` y g es `notasAlumno`
Para que la composición f . g se pueda realizar el dominio de f tiene que incluir a la imagen de g (cosa que sucede)

```
estaAprobado unAlumno = (todosMayoresOIgualesQueCuatro.notasAlumnos) unAlumno
```

Volviendo ...

```
nombresAlumnosAprobados unCurso = map nombreAlumno (filter estaAprobado unCurso)
```

Primero obtenemos una lista con los alumnos aprobados (una lista de tuplas) y despues obtenemos una lista de strings (lista de nombres) aplicando la "transformación" `nombreAlumno` a cada elemento de la lista de alumnos aprobados.

Una forma de escribir con notación mátematica la función `nombreAlumnosAprobados` podría ser

```
nombreAlumnosAprobados unCurso = { nombreAlumno x / x ∈ unCurso ∧ estaAprobado x }
```

Pasar esto a Haskell es trivial

```
nombresAlumnosAprobados unCurso = [ nombreAlumno x | x ← unCurso , estaAprobado x ]
```

A está forma de denotar listas la vamos a llamar, **listas por comprensión** y tienen la siguiente sintaxis

["transformación" sobre cada x | x ← lista , condición sobre cada x]

Podemos definir map y filter con listas por comprensión (está no es la forma en que realmente están definidas)

```
filter funcion lista = [ x | x ← lista , funcion x ]

map funcion lista = [ funcion x | x ← lista ]
```

Qué pasa si ahora queremos conocer una tupla que tiene el nombre del alumno y su promedio de notas pero solo de los alumnos aprobados de un curso

```
-- promedioAprobados :: [Alumno] -> [(Nombre,NotaPromedio)]
promedioAprobados curso = [(nombreAlumno x , promedioAlumno x)| x <-curso ,
estaAprobado x]
promedioAlumno unAlumno = promedio (notas unAlumno)

--Si queremos el promedio entero
promedio numeros = div (sum numeros) (length numeros)
-- O bien
promedio numeros = fromIntegral (sum numeros) / fromIntegral (length numeros)
```

Y ahora?

En todos los ejemplos que hicimos hasta ahora utilizamos tuplas de 2 elementos, pero qué pasa si quiero usar más elementos en una tupla?

Las funciones que ya vienen en Haskell para manejar tuplas son fst, snd, swap ... pero cuál es su dominio e imagen?

```
fst :: (a,b) -> a
snd :: (a,b) -> b
swap :: (a,b) -> (b,a)
```

El dominio de todas estás funciones es **Tuplas de 2 elementos**

Como vimos anteriormente cuando hablamos de un tipo `a` o `b`, nos estamos refiriendo a una **variable de tipo**, dicha variable puede ser reemplazada por cualquier tipo (respetando que todas las apariciones de `a` o `b` en la definición serán reemplazadas por ese tipo).

Hagamos un ejemplo:

```
nombreAlumno = fst
```

```
notasAlumno = snd

cursoK9 = [("mica",[8,6,9,10,8]),("pepa",[2,1,3]),("caro",[6,7,9]),("beto",[10,9,2,8])]

cantidadNotas unAlumno = length (notasAlumno unAlumno)

quienesDieronParcialito curso = filter ((3<).cantidadNotas) curso

nombres unCurso = map nombreAlumno unCurso

nombresAlumnosAprobados unCurso = map nombreAlumno (filter estaAprobado unCurso)
```

Se puede utilizar la función nombreAlumno solo con tuplas de 2 elementos!

```
> nombreAlumno ("Gauss", [10,10,10,10])
"Gauss"
```

Si queremos extender nuestro ejemplo de cursos y alumnos, podemos intentar que un alumno sea más que su nombre y sus notas (qué poético ...).

Podemos ver a un alumno como, un nombre, una lista de notas y su .... número de legajo! (o.0)

```
> nombreAlumno ("Gauss", 1179321, [10,10,10,10])
ERROR
```

El dominio de la función `nombreAlumno` es igual al dominio de la función `fst`, **tuplas de 2 elementos**, por lo que no va a funcionar para nuestra nueva forma de representar un alumno

Entonces cómo le hacemos ???

Pattern-Matching

1. El problema lo vamos a tener en las funciones `nombreAlumno` y `notasAlumno`
2. No queremos modificar ninguna de las otras funciones que definimos en nuestro programa

Cómo está definida la función fst y snd?

```
fst (x,y) = x
snd (x,y) = y
```

Nota: a pesar de no conocer estas definiciones cuando teníamos a nuestro alumno representado solo por su nombre y sus notas no hicimos

```
-- ESTO ESTÁ MAL !!! (funciona, pero está CONCEPTUALMENTE MAL)
notasAlumno (nombre,notas) = snd (nombre,notas)
```

En el ejemplo anterior queremos que se encargue snd de resolver el problema, no me interesan los valores que componen a un alumno

```
notasAlumno unAlumno = snd unAlumno

-- o bien
notasAlumno = snd

-- o bien
notasAlumno (nombre,notas) = notas
```

Volviendo, podemos usar esta misma idea para definir nuestras funciones que van a manejar alumnos como tuplas de 3 elementos

```
notasAlumno (nombre, legajo, notas) = notas
```

Qué es una estructura de datos?

Entre varias cosas una estructura de datos nos da la oportunidad de tratar a un conjunto de datos como

- un solo elemento
- los elementos que lo componen

Ejemplo

Cuando hablamos de un alumno en la función `cantidadNotas` nos interesa ver al alumno como UN solo elemento

```
-- A pesar de que representamos a un alumno como una tupla lo vemos como una sola variable
cantidadNotas unAlumno = length (notasAlumno unAlumno)
```

Cuando hablamos de un alumno en la función `nombreAlumno` nos interesa ver los elementos que componen a un alumno (en particular sus notas)

```
-- Como representamos a un alumno como una tupla usando pattern-matching podemos usar distintas variables para conocer a sus elementos
nombreAlumno (nombre, legajo, notas) = nombre
```

Unificación y Pattern-Matching

En clases pasadas dijimos que dentro de los conceptos del paradigma funcional no estaba incluido el concepto de **asignación**. Para dejar en claro esto vamos a llamar a esta idea **Asignación Destructiva**, esto se debe a que una asignación me permite "destruir" el valor que tiene una variable en un momento dado y reemplazarlo por otro.

Las variables en el paradigma funcional se asemejan a las variables en matemática, i.e. cuando una variable toma un valor el valor de la variable no va a cambiar. Esta idea se contrapone a la idea de variable en el paradigma procedural, en donde se ve una variable como una posición de memoria y cuyo valor puede ser reemplazado una y otra vez durante mi programa.

Entonces, esta idea de asignación no tiene sentido cuando pensamos en variables matemáticas (si en un momento dado decimos que una variable X vale 1 su valor será 1 y ningún otro hasta que se empiece a buscar otra soluciòn al mismo sistema, en ese momento se desligan todas las variables y se empieza de nuevo). Las variables en el paradigma funcional se asemejan a la idea de variable matemática, y el mecanismo por el cual se le dan valores a las variables se llama **unificación**.

Se dice que 2 términos unifican si existe algún reemplazo de todas las variables (de los 2 términos) que haga a los términos iguales.

Y pattern-matching?

Bueno, la diferencia entre decir **pattern-matching** y **unificación** es bastante gris (algunos autores lo consideran sinónimos). Es muy común decir "unifica" o "matchea" indistintamente.

Mayormente vamos a hablar de **pattern-matching** en la unificación de valores compuestos (tuplas y listas - ya que estos valores tienen un patrón definido) y vamos a hablar de **unificación** en los demás casos (más sobre esto cuando veamos Lógico).

As pattern

Volvamos al ejemplo donde queremos obtener los nombres de los alumnos aprobados (considerando a un alumno como una tupla de 3 elementos)

```
nombresAlumnosAprobados unCurso = map nombreAlumno (filter estaAprobado unCurso)
```

Si queremos resolver esto con listas por comprensión podemos hacer

```
nombresAlumnosAprobados unCurso = [ nombreAlumno unAlumno | unAlumno ← unCurso , estaAprobado unAlumno ]
```

Ahora bien, en vez de usar la función `nombreAlumno` podemos usar pattern-matching para obtener el nombre de `unAlumno`

```
nombresAlumnosAprobados' unCurso = [ nombre | (nombre,legajo,notas) ← unCurso , estaAprobado (nombre,legajo,notas) ]
```

Lo molesto de esta solución es que a la función `estaAprobado` le estamos enviando todo el alumno (toda la tupla) pero tuvimos que volver a construirla a partir de sus componentes.

Se puede usar el "as-pattern" para aplicar 2 patrones distintos a un mismo valor, en este ejemplo podemos ver a cada alumno como

- `unAlumno` (el alumno completo)
- `(nombre,legajo,notas)` (las componentes del alumno)

El "as-pattern" se aplica usando el símbolo `@`

```
nombresAlumnosAprobados'' unCurso = [ nombre | (nombre,legajo,notas)@unAlumno ← unCurso , estaAprobado unAlumno ]
```

Variable anónima

Cuando no nos interesa un valor, en su lugar en vez de escribir "una variable" podemos escribir _ (guión bajo).

A la variable _ la llamamos, **variable anónima**.

Ejemplo:

```
nombresAlumnosAprobados'' unCurso = [ nombre | (nombre,_,_)@unAlumno ← unCurso , estaAprobado unAlumno ]
```

Cuidado, en esta versión

```
nombresAlumnosAprobados' unCurso = [ nombre | (nombre,legajo,notas) ← unCurso , estaAprobado (nombre,legajo,notas) ]
```

no podemos usar la variable anónima, porque nada nos garantiza que las primeras 2 variables anónima (las rojas) sean iguales a las 2 variables anónimas siguientes (las verdes). El decir *anónimo* significa que su valor no es importante para la definición y en este caso si lo es!

```
-- esto está mal!
nombresAlumnosAprobados' unCurso = [ nombre | (nombre,_,_) ← unCurso , estaAprobado (nombre,_,_) ]
```

Más Pattern Matching

```
-- Posible definición de la función not
not :: Bool -> Bool
not True = False
not False = True
```

Como el matcheo se hace en orden secuencial, en la segunda línea el parámetro se puede reemplazar

por la variable anónima

```
-- Posible definición de la función not
not :: Bool -> Bool
not True = False
not _ = True
```

Patrones con tuplas

```
fst :: (a, b) → a
fst (x , _ ) = x

snd :: (a, b) → b
snd ( _ , y) = y

swap :: (a, b) → (b, a)
swap ( x , y) = ( y , x)
```

Patrones con listas

```
null :: [a] → Bool
--Es como el isEmpty
null [ ] = True
null (_:_) = False

head :: [a] → a
head (x :_) = x

tail :: [a] → [a]
tail (_: xs) = xs
```

El patrón de cabeza:cola tiene que estar entre paréntesis por el tema de la precedencia de la aplicación.

P.ej.
```
tail _ : xs = xs
```

sin paréntesis significa
```
(tail _) : xs = xs
```

lo cual NO funciona

Patrón de enteros

```
anterior :: Int → Int
```

En vez de

```
anterior 1 = 0
anterior n = n - 1
```

Podemos escribir

```
anterior 1 = 0
anterior (n + 1) = n
```

Ejemplos locos

```
factores :: Int → [Int ]
factores n = [x | x ← [1 .. n ], mod n x == 0]

esPrimo :: Int → Bool
esPrimo n = factores n == [1,n]
```

-- zip es una función que ya viene en Haskell
-- recibe 2 listas y devuelve una lista de tuplas donde cada componente pertenece a una de esas 2 listas
-- la lista de tuplas tiene el tamaño de la menor de las 2 listas enviadas

> zip [’a’, ’b’, ’c’] [1, 2, 3, 4]
[(’a’, 1), (’b’, 2), (’c’, 3)]

> zip "chau" "hola"
[('c','h'),('h','o'),('a','l'),('u','a')]

```
deAPares :: [a] → [(a, a)]
deAPares xs = zip xs (tail xs)
```

> deAPares [1, 2, 3, 4]
[(1, 2), (2, 3), (3, 4)]

```
estaOrdenada :: [a] → Bool
estaOrdenada xs = and [x <= y | (x , y) ← deAPares xs ]
```

-- and es una función que ya viene en Haskell
-- Recibe una lista de booleanos y devuelve True si todos esos booleanos son True; False en cualquier otro caso