Contenido

1 Tuplas

Permite representar un tipo de dato compuesto, pero con elementos de distinto tipo. El número de elementos es fijo (siempre el mismo).

```
(23, 02, 1973)
("Juan", 23)
```

Una tupla es un tipo de dato compuesto donde
$(x_1, x_2.. x_n)$ tiene tipos $t_1, t_2... t_n$

Algunos ejemplos de tuplas: ¿De qué tipo son?

Tupla	Tipo de la tupla
`(1, [2], 3)`	`(Int, [Int], Int)`
`('a', False)`	`(Char, Bool)`
`((1,2),(3,4))`	`((Int, Int), (Int, Int))`

1.1 Comparación entre tuplas y listas

Comparamos tuplas con listas:

- Las listas requieren que todos los elementos sean homogéneos: no podemos mezclar en una misma lista números y strings.
- El número de elementos de una lista es variable, puede ser infinito.
- La lista es un tipo de dato recursivo, la tupla no, aunque ambos son compuestos.

Algunas funciones para hacer en el pizarrón en conjunto:

```
sumaPar (a, c) (b, d) = (a+b, c+d)
minPar (a,b) = min a b
```

Hay funciones estándar para separar una tupla de dos elementos:

```
fst (a, _) = a
snd (_, b) = b
```

¿De qué tipo son?

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

Si persona es una dupla, podemos definir:

```
fernando = ("Fernando", 43)
nombre = fst
edad = snd
```

Entonces tenemos un nombre de función mucho más representativo de lo que hace (lo que en alguna bibliografía se conoce como *intention revealing*):

```
>nombre Fernando
"Fernando"
```

Pero no es la única forma de agrupar información de una entidad. A continuación veremos otras opciones.

2 Tipos de dato definidos por el usuario

```
data Persona = Persona String Int
```

Esto se lee como: “el tipo de dato Persona utiliza un constructor –de nombre Persona- que recibe un String y un Int”.

De esta manera podemos definir las funciones que permiten conocer el nombre y la edad:

```
nombre (Persona n e) = n
edad (Persona n e) = e
```

Tener en cuenta la línea de la definición “data”, más arriba. La sintaxis de una persona no requiere paréntesis pero, al igual que pasaba con las listas, para decir “todo esto es un único parámetro” vamos a necesitar usarlos.

Lo evaluamos en Haskell:

```
>nombre (Persona "Miguel" 34)
"Miguel"
```

¿Qué diferencias hay entre modelar una persona como una tupla o como un tipo de dato?
¿Qué tipo espera la función nombre?

```
nombre :: Persona -> [Char]
```

Una persona, mientras que la definición de nombre = fst nos da

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

Y relacionado con esto: la tupla ofrece una solución general que sirve para modelar personas,

empleados y muchas otras cosas. Justamente por eso no es intuitiva, la estructura tiene que ser conocida por quien la usa: tengo una tupla con dos elementos String e Int, no es fácil darse cuenta de que se trata de una estructura que modela una persona, donde el primer elemento se asocia al nombre y el segundo elemento a la edad.

Para invocar a la función nombre, necesitamos construir primero un valor de tipo Persona, esto lo hacemos mediante el constructor Persona:

```
>:t Persona "Miguel" 34
Persona "Miguel" 34 :: Persona
```

De hecho la evaluación parcial funciona perfectamente:

```
> :t Persona "Miguel"
Persona "Miguel" :: Int -> Persona
```

Es decir, Persona “Miguel” es una función que espera la edad y devuelve una persona.

2.1 Perros y personas

Ahora queremos modelar un perro. El perro tiene un nombre, pero no nos interesa conocerlo. Sí queremos saber qué edad tiene y si es de pedigree:

```
data Perro = Perro Int Bool
```

Recordemos, el primer Perro hace referencia al nombre del tipo de dato.
El segundo al nombre que recibe el constructor de ese tipo de dato.

Tenemos a

```
firulais = Perro 5 False
```

Podemos armar una función para conocer la edad de firulais:

```
edad (Perro edad _ ) = edad
```

Pero Haskell no nos deja… porque ya definimos una función edad cuyo primer argumento es una persona. Claro, todo esto nos fuerza a replantear un poco las cosas:

¿Podríamos usar tuplas en lugar de los tipos de dato definidos por el usuario? Mmm… eso es un problema porque la persona es una tupla que guarda nombre y edad, y el perro edad y pedigree, por más que movamos la edad al segundo lugar, claramente no podemos forzar que se integren un String con un Bool.

¿Qué es lo que estamos queriendo hacer?

Permitir que la función edad acepte tanto perros como personas. Necesitamos que en un determinado contexto los perros y las personas se puedan intercambiar. Si el lenguaje tiene chequeo estático de tipos, como Haskell, entonces necesitamos que tanto perros como

personas tengan un tipo en común: la función edad no puede recibir cualquier `a`, ni una tupla cualquiera (eso sería polimorfismo paramétrico, como en el caso de `head/1` o `length/1`). Edad necesita un perro, o una persona, o algo a lo que le pueda preguntar la edad (lo que forma un ejemplo de *polimorfismo ad-hoc*, hay restricciones en los tipos de los parámetros a recibir). Entonces tenemos que replantear el tipo Persona, vamos a crear un tipo nuevo que acepte tanto perros como personas:

```
data SerVivo = Persona String Int | Perro Int Bool
```

Leemos: el tipo de dato SerVivo tiene dos constructores:

- Persona, que necesita un String (nombre) e Int (edad)
- y Perro, que necesita un Int (edad) y un Bool (si tiene pedigree).

La función edad la vamos a trabajar con pattern matching:

```
edad (Persona _ edad) = edad
edad (Perro edad _) = edad
```

Y ahora sí lo evaluamos:

```
>edad firulais
5
```

Podemos tener en una misma lista perros y personas, y filtrar los viejos:

- Una persona de más de 60 años es vieja
- La sabiduría popular canina dice que un año perruno equivale a 7 años humanos, entonces un perro cuya edad * 7 > 60 se puede considerar viejo

Lo que difiere es la edad “real” considerable para cada ser vivo. Resolvemos esto nuevamente mediante pattern matching:

```
edadReal (Persona _ edad) = edad
edadReal (Perro edad _) = edad * 7
```

La función viejo sabe que hay que decidir en base a la edad real (eso en mi pueblo se llama delegar):

```
viejo serVivo = edadReal serVivo > 60
```

Preguntamos si firulais es viejo:

```
> viejo firulais
```

`False` (porque 5 * 7= 35 < 60)

Preguntamos si Miguel es viejo:

```
> viejo (Persona "Miguel" 34)
```

`False` (porque 34 < 60)

La función viejo, ¿necesita distinguir perros de personas?
No, y eso es lo bueno…
Tarea para el lector: añada un constructor Tortuga para el tipo SerVivo. De la tortuga

conocemos su edad y si viven en capital. Las tortugas de Capital son viejas a partir de los 50 años (por el stress), las que no son viejas a partir de los 70 años. ¿Qué modificaciones tiene que introducir en el ejemplo que planteamos? ¿Qué ocurrió con la función viejo, sufrió algún impacto?

2.2 Otro ejemplo: Empleados

Queremos calcular el sueldo de los empleados de nuestra empresa. Tenemos dos tipos de empleado:

- Los comunes: nos interesa saber el sueldo básico y el nombre.
- Los jerárquicos: nos interesa conocer el sueldo básico, la cantidad de gente a cargo y el nombre.

El sueldo que cobran los comunes se determina por el sueldo básico, en los empleados jerárquicos se calcula como sueldo básico + plus por la cantidad de gente a cargo (1.000 si tiene más de 10 empleados a cargo o 300 en caso contrario).

Primero definimos el tipo de dato Empleado, con dos constructores posibles:

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

A continuación, definimos un juego de datos:

```
juan = Comun 4000 "juan"
hugo = Jefe 6000 2 "hugo"

empleados = [juan , hugo, Comun 2500 "braulio", Comun 2500 "juliana"]
```

Ahora queremos saber el sueldo de un empleado. La función sueldo va a recibir un Empleado y va a devolver… su sueldo:

```
sueldoDe (Comun sueldo _ ) = sueldo
sueldoDe (Jefe sueldo genteACargo _ ) =
      sueldo + plusPorGenteACargo genteACargo
```

¿Estamos definiendo la función para dos tipos distintos?
NO. El tipo sigue siendo Empleado, pero un Empleado puede tener dos formas distintas... ¡lo mismo que pasaba con las listas! ¿Se acuerdan de esto?

```
length [] = 0
length (_:xs) = 1 + length xs
```

Entonces: el tipo es uno solo, pero usamos **pattern matching** para diferenciar formas dentro de un mismo tipo.

Desarrollamos el plus por gente a cargo:

```
plusPorGenteACargo cantidad | cantidad > 10 = 1000
                            | otherwise    = 300
```

Podemos obtener la lista de sueldos de los empleados:

```
totalSueldos [] = 0
totalSueldos (emp:emps) = sueldoDe emp + totalSueldos emps
```

Ojo, si yo comento el sueldo del empleado común `totalSueldos empleados`, WinHugs devuelve:

```
Program error: {sueldoDe juan}
```

Es algo para destacar que, pese a ser un lenguaje con chequeo estático, con la solución que construimos, Haskell no nos puede garantizar que todos los empleados definan la función `sueldoDe`. No hay errores en tiempo de compilación, el error es de **pattern matching** en tiempo de ejecución. Esto se ve algo más claro si estamos usando GHC. El error nos dice:

```
*** Exception: test.hs:...: Non-exhaustive patterns in function sueldoDe
```

El problema es el mismo que si intento pedirle al motor el head de una lista vacía[1]: Es correcto usar head con listas, es por eso que no hay error de tipos, pero no tiene sentido pedirlo para la lista vacía, y por lo tanto no está definido.

3 Tipos y clases

¿Podemos preguntar si un empleado es igual a otro?

```
>juan (==) juan
ERROR - Illegal Haskell 98 class constraint in inferred type
*** Expression : juan == juan
*** Type       : Eq Empleado => Bool
```

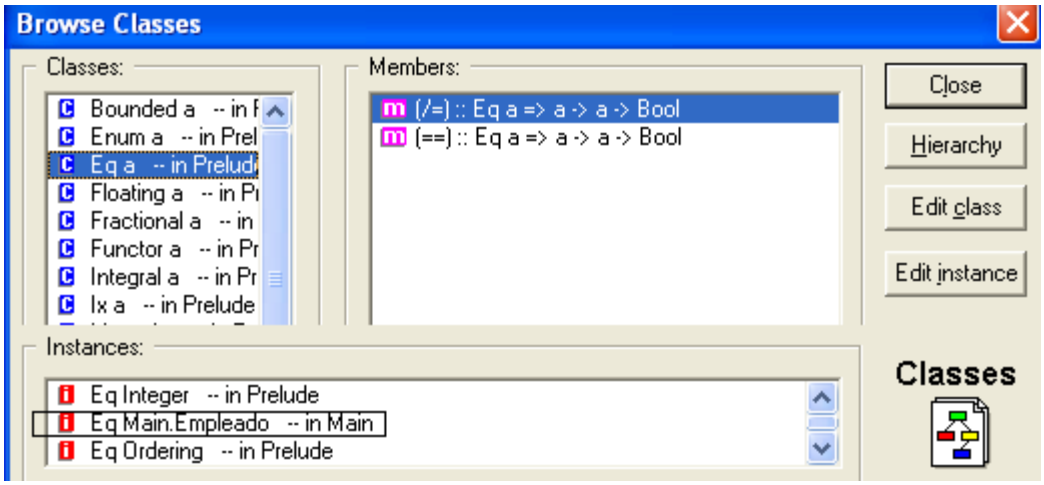
No, al tratar de aplicar el operador (==) con dos empleados, la definición no matchea:

```
>:t (==)
(==) :: Eq a => a -> a -> Bool
```

Un tipo Empleado no es un Eq, entonces no se cumple la restricción de la clase Eq (Illegal class constraint) cuando la quiere asociar al empleado (el tipo inferido, inferred type).

De la misma manera que en Java una clase puede implementar muchas interfaces, un tipo de dato puede pertenecer a muchas clases. En este caso vamos a hacer que el tipo Empleado sea de la clase Eq:

[1] GHC nos muestra un mensaje de error distinto, específico para este caso (`*** Exception: Prelude.head: empty list`), pero el problema de fondo sigue siendo de pattern-matching.



Primero tenemos que decir que el tipo empleado deriva la clase Eq:

```
data Empleado = Comun Float String | Jefe Float Int String deriving Eq
```
[2]

Ahora tenemos que encontrar una definición de negocio que determine cuándo dos empleados son iguales:

Por nombre (si se llaman igual, son iguales)

```
instance Eq Empleado where
   (==) unEmpleado otroEmpleado =
        nombre unEmpleado == nombre otroEmpleado
```

Por sueldo (si ganan lo mismo, son iguales)

```
instance Eq Empleado where
   (==) unEmpleado otroEmpleado =
        sueldoDe unEmpleado == sueldoDe otroEmpleado
```

El tipo es una instancia de la clase, y define el comportamiento para el operador (==).

Hay que tener en cuenta que este es un ejemplo didáctico, no es de ninguna manera una recomendación (de hecho, es una mala práctica en el paradigma OO: no debería haber dos objetos empleado con el mismo nombre, o dos personas con el mismo documento de identidad).

[2] Mientras Haskell 98 permite que los tipos deriven de una lista acotada de clases “built-in” (Eq, Ord, Enum, Bounded, Show y Read), GHC extiende esta lista. Para más información véase http://en.wikibooks.org/wiki/Haskell/Class_declarations y http://www.haskell.org/ghc/docs/7.6.2/html/users_guide/deriving.html

Evaluamos ahora, en base al criterio de igualdad por nombre:

```
> Comun 1000 "pepe" == Jefe 1000 2 "pepe"
True
```

También funciona la búsqueda con elem:

```
>elem (Comun 2 "juan") empleados
```

`True`, porque hay un empleado de nombre juan en esa lista.

Aquí lo que nos importa no es tanto el hecho de escribir una función igualdad para empleados, sino cómo esa función encaja en otras funciones que ya vinimos usando, como elem:

```
>:t elem
elem :: Eq a => a -> [a] -> Bool
```

elem necesita un Eq (y luego una lista de Eqs). Cuando definieron la función elem, no existía el tipo Empleado, pero basta con que un Empleado defina la igualdad para que elem lo acepte:

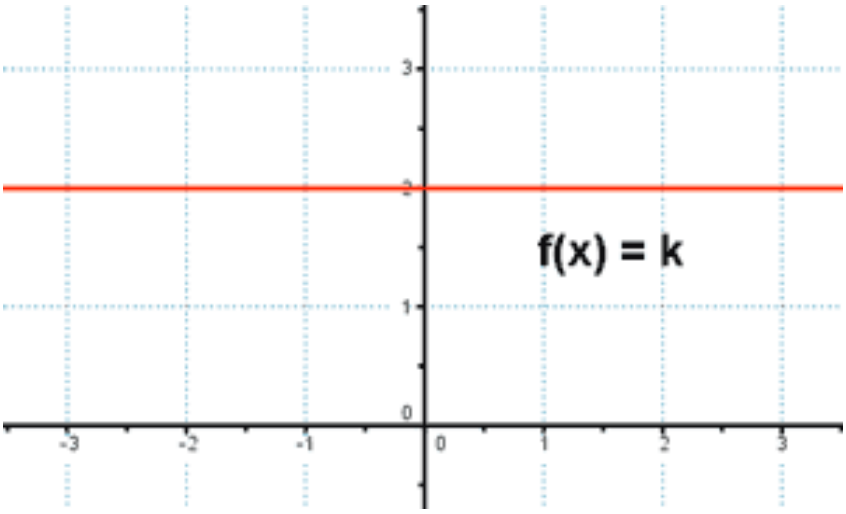
4 Funciones constantes

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 o de tipos definidos por el usuario. *Ejemplo:* 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.