Paradigma Funcional - Clase 4

Repaso

Dado el siguiente programa

```
fichasJugadas = [(6,6),(6,4),(4,2),(2,2),(2,5)]
```

- Hacer una función que reciba una lista de fichas de dominio y me devuelva una nueva lista con los extremos

```
> extremos fichasJugadas
[6,5]
```

- Hacer una función que reciba una ficha y me dice si puedo jugarla

```
> sePuedeJugar (3,2) fichasJugadas
False

> sePuedeJugar (1,5) fichasJugadas
True
```

- Hacer una función que reciba una lista y un elemento A y retorne un elemento B que es el elemento que está justo delante en la lista enviada como parámetro

```
> anterior "hola" 'a'
'l'

> anterior [1,384,83,310] 384
1
```

- Hacer una función que me diga si una lista de fichas está bien armada. Esto se cumple si para todos los elementos de la lista de fichas (excepto el primero) su ficha anterior tiene por extremo derecho el mismo valor que el extremo izquierdo de la ficha que le sigue.

```
> estaBienArmada fichasJugadas
True

> estaBienArmada [(2,1),(2,2),(2,5)]
False
```

- Tip: existe una función `findIndices :: (a -> Bool) -> [a] -> [Int]` que ya viene en Haskell

```
> findIndices (=='h') "hola"
[0]

> findIndices (=='l') "hola"
[2]

> findIndices (=='a') "saraza"
```

```
[1,3,5]
```

Conceptos Básicos

Para nosotros, en funcional, un tipo es el domino o imagen de alguna función.

Por ejemplo, el tipo `Bool` es el dominio e imagen de la función `not` y el tipo `Bool` contiene los valores `True` y `False`.

```
False :: Bool
True :: Bool
not :: Bool → Bool
```

Vale la pena recordar que incluso expresiones sin evaluar tienen un tipo

```
not False :: Bool
not True :: Bool
not (not False) :: Bool
```

En Haskell, todas las expresiones deben tener un tipo y ese tipo se "calcula" antes de la evaluación de la expresión a través de un proceso llamado *Inferencia de Tipos*.

Inferencia de Tipos

La clave de este proceso es la regla por la cual se obtiene el tipo de la aplicación de una función:

$$\frac{f :: A \to B \qquad e :: A}{f\,e :: B}$$

En términos más mundanos, lo que la regla dice es

1. si `f` es una función cuyo dominio es `A` e imagen es `B` y,
2. si `e` es una expresión de tipo `A`
3. entonces, el tipo de `f e` (usamos a `e` como argumento de `f`) es `B`

Ejemplo, el tipo de `not False` se puede inferir con esta regla

1. si `not` es una función cuyo dominio es `Bool` e imagen es **`Bool`** y,
2. si `False` es una expresión de tipo `Bool`
3. entonces, el tipo de `not False` es **`Bool`**

Otro ejemplo, si queremos saber de que tipo es la expresión `not 3` vemos que no tiene un tipo siguiendo la regla que especificamos anteriormente, porque requeriría que `3` sea de tipo `Bool` pero `3` no es un booleano.

Expresiones como `not 3, 2 + True, length (4,3), fst [1,2]`, etc. no tienen un tipo y por lo tanto no son expresiones válidas en Haskell.

Como la inferencia de tipos es un proceso ANTERIOR a la evaluación, los programas que hacemos en Haskell son Type Safe (más de esto en próximas clases).

Tiene tipo ≠ Bien

Que una expresión tenga tipo no significa que sea una expresión libre de errores.

```
head :: [ a ] -> a
head [] :: a
```

Pero a pesar de esto, si evaluamos la expresión

```
> head []
```
Error

Una consecuencia de la inferencia es que expresiones que tienen sentido no tienen tipo y por lo tanto no son expresiones válidas en Haskell.

Por ejemplo

```
funcionLoca x
     | x > 0 = 1
     | otherwise = True
```

Como por una rama retornamos un Int (`1 :: Int`) y por la otra retornamos un Bool (`True :: Bool`) no se puede obtener la imagen de la función `funcionLoca`, ergo la `funcionLoca` no se puede escribir en Haskell porque no tiene un tipo.

Recuerden que en Haskell ustedes pueden obtener el tipo de cualquier expresión escribiendo :t (type) en el interprete

```
> :t not
not :: Bool -> Bool

> :t not False
not False :: Bool

> :t not 3
```
Error

Clasificando tipos

Vamos a jugar un poco con algunas funciones que ya vienen en Haskell

La función "mas"

```
-- Ejemplo 1 (suena razonable sumar dos números enteros)
> 2 + 3
6

-- Ejemplo 2 (suena razonable sumar dos números float)
> 2.8 + 3.1
5.9

-- Ejemplo 3 (suena razonable sumar dos números fractional)
```

```
> (2/3) + (1/3)
1.0
```

Si tuvieramos que decir el dominio e imagen de la función `(+)` tendríamos uno específico para cada ejemplo

-- Ejemplo 1
```
(+) :: Int -> Int -> Int
```

-- Ejemplo 2
```
(+) :: Float -> Float -> Float
```

-- Ejemplo 3
```
(>) :: Fractional -> Fractional -> Fractional
```

Pero dijimos que una función solo puede tener un tipo, o sea, solo puede tener un dominio e imagen determinado.

Ahora bien, podemos decir que el tipo de `(+)` es

--Esto está mal
```
(+) :: a - > a -> a
```

Pero tampoco nos sirve, al no decir nada sobre el tipo `a` la función `(+)` podría recibir como parámetro cualquier cosa:

```
True :: Bool
False :: Bool
```
Cuando uso `(+)` el tipo `a` sería `Bool`

-- no tiene sentido sumar booleanos con la función `(+)`
```
> True + False
```
Error

```
last :: [b] -> b
head :: [b] -> b
```
Cuando uso `(+)` el tipo `a` sería `([b] -> b)`

-- no tiene sentido sumar funciones con la función `(+)`
```
> last + head
```
Error

La función "mayor"

-- Ejemplo 1 (suena razonable preguntar si un número es mayor que otro)
```
> 1 > 3
False
```

-- Ejemplo 2 (una palabra es más grande que otra si "aparece más cerca del final en un diccionario")
```
> "hola" > "chau"
True
```

-- Ejemplo 3 (una tupla de 2 elementos que son números se puede ver como un punto

2D, es más grande mientras este en el primer cuadrante lo más alejado del origen de coordenadas)

```
> (3,2) > (0,0)
True
```

Si tuvieramos que decir el dominio e imagen de la función `(>)` tendríamos uno específico para cada ejemplo

```
-- Ejemplo 1
(>) :: Int -> Int -> Bool

-- Ejemplo 2
(>) :: String -> String -> Bool

-- Ejemplo 3
(>) :: (Int,Int) -> (Int,Int) -> Bool
```

Pero dijimos que una función solo puede tener un tipo, o sea, solo puede tener un dominio e imagen determinado.

Ahora bien, podemos decir que el tipo de `(>)` es

```
-- Esto está mal
(>) :: a - > a -> Bool
```

Pero tampoco nos sirve, al no decir nada sobre el tipo `a` la función `(>)` podría recibir como parámetro cualquier cosa:

```
True :: Bool
False :: Bool
```

Cuando uso `(>)` el tipo `a` sería `Bool`

```
-- "no tiene sentido" comparar booleanos con la función (>)
> True > False
Error
```

```
last :: [b] -> b
head :: [b] -> b
```

Cuando uso `(>)` el tipo `a` sería `([b] -> b)`

```
-- no tiene sentido comparar funciones con la función (>)
> last > head
Error
```

La función "igual"

```
-- Ejemplo 1 (podemos ver si 2 números son iguales)
> 1 == 3
False

-- Ejemplo 2 (dos string son iguales si coinciden en todos sus elementos)
> "hola" == "chau"
False
```

-- Ejemplo 3 (podemos comparar tuplas por igualdad)
```
> (3,2) == (2+1,2)
True
```

Si tuvieramos que decir el dominio e imagen de la función `(==)` tendríamos uno específico para cada ejemplo

-- Ejemplo 1
```
(==) :: Int -> Int -> Bool
```

-- Ejemplo 2
```
(==) :: String -> String -> Bool
```

-- Ejemplo 3
```
(==) :: (Int,Int) -> (Int,Int) -> Bool
```

Pero dijimos que una función solo puede tener un tipo, o sea, solo puede tener un dominio e imagen determinado.

Ahora bien, podemos decir que el tipo de `(==)` es

-- Esto está mal
```
(==) :: a - > a -> Bool
```

Pero tampoco nos sirve, al no decir nada sobre el tipo `a` la función `(==)` podría recibir como parámetro cualquier cosa:

```
last :: [b] -> b
head :: [b] -> b
```
Cuando uso `(==)` el tipo `a` sería `([b] -> b)`

-- no tiene sentido comparar funciones con la función `(==)`
```
> last == head
```
Error

Restricciones de tipo

Nos gustaría poder definir los siguientes tipos (a.k.a dominios e imagenes)

`(+)` :: (Si asumimos que `a` es "**num**érico") entonces `a - > a -> a`

`(>)` :: (Si asumimos que `a` es "**ord**enable") entonces `a - > a -> Bool`

`(==)` :: (Si asumimos que `a` es "**eq**uiparable") entonces `a - > a -> Bool`

En Haskell eso se escribe de la siguiente manera

```
(+) :: (Num a) => a - > a -> a
```

```
(>) :: (Ord a) => a - > a -> Bool
```

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

`Num, Ord y Eq` son restricciones de tipo, en Haskell se las conoce como **Typeclasses** (no confundan esto con el término Clase que usamos en el paradigma de objetos!!)

En cada Typeclass se definen un conjunto de funciones que los tipos pertenecientes deben implementar

`Num a`

- `(+), (-), (*) :: a -> a -> a`
- `negate, abs, signum :: a -> a`
- etc.

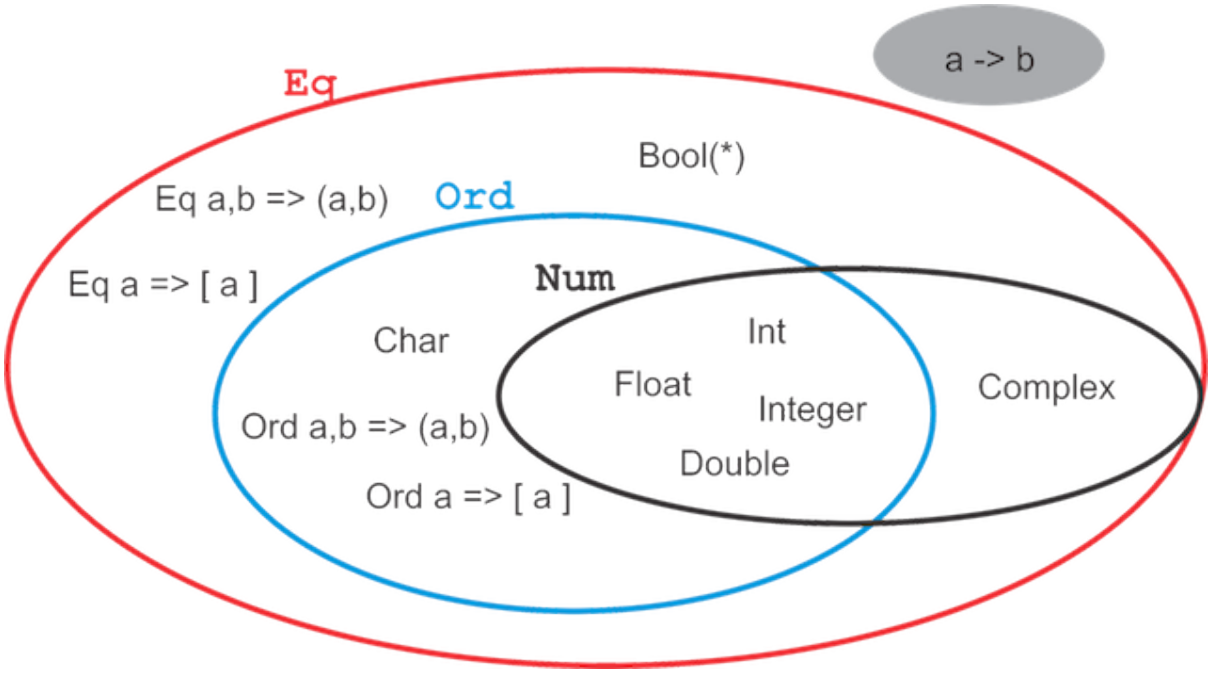
`Ord a`

- `(<), (<=), (>=), (>) :: a -> a -> Bool`
- `max, min :: a -> a -> a`
- etc.

`Eq a`

- `(==) :: a -> a -> Bool`
- `(/=) :: a -> a -> Bool`

Qué tipos pertenecen a cada restricción?



`a->b` (Tipo función) no pertenece a ninguna clase
Recordar que `String` es sinónimo de `[Char]` por ende está incluido en `(Ord a) => [ a ]`
(*) el tipo `Bool` en realidad pertenece a `Ord` (`True` es mayor que `False`)

Ejemplos:

```
elem :: (Eq a) => a -> [ a ] -> Bool
elem unElemento unaLista = any (unElemento==) unaLista

max :: (Ord a) => a -> a -> a
```

```
max x y
    | x > y = x
    | otherwise = y

sum :: (Num a) => [ a ] -> a
sum numeros = <LO VAMOS A RESOLVER MÁS ADELANTE>
```

La restricción Show

Cuando utilizamos el interprete, los resultados de nuestras funciones son valores (valores simples, compuestos o funciones), para que se puedan mostrar por pantalla esos valores tienen que tener una representación en forma de cadena de caracteres (`String`).

La magia la realiza una función llamada `show`

```
> show 3
"3"

> show True
"True"

> show (2,3)
"(2,3)"
```

Ahora bien, no todos los valores pueden ser parámetro de la función `show`. Por ejemplo **las funciones no tienen una representación en String**

```
> show length
Error
```

Debido a esto, cuando quieren mostrar por pantalla una lista de funciones, una función parcialmente aplicada, o en general una función que retorna una función les va a tirar un error

```
> [fst,snd]
Error

> (.)
Error

> (3+)
Error
```

Veamos el tipo de la función show

```
show :: (Show a) => a -> String
```

La única función que se define en la restricción Show es la función show

Qué tipos pertenecen a la restricción Show?

- `Bool`
- `Char`
- `Double`
- `Float`
- `Int`
- `Integer`
- `(Show a) => [ a ]` --Listas
- `(Show a,b) => (a,b)` --Tuplas

Resumiendo: casi todos todos los tipos menos el tipo función

Tuplas tuplas tuplas

Hasta ahora cuando teníamos que representar un valor, compuesto por otros valores de tipos distintos, usábamos Tuplas.

-- Vamos a hacer un ejemplo con tuplas de 2 elementos por simplicidad, pero lo mismo se aplica para tuplas de n elementos

-- Si queremos representar un alumno por su nombre (un String) y sus notas (una lista de Int = [Int])

```
fede = ("Federico",[2,3])
lider = ("Líder",[10,10,10,10,10])
ger = ("Germain",[8,9,10])

cursoK9 = [fede,lider,ger]

empezaronMal unosAlumnos = filter ((4>).head.notasAlumno) unosAlumnos
```

-- Si queremos representar una película por su título (unString) y los puntajes que le ponen los críticos en imd (una lista de Int = [Int])

```
narnia = ("Pedornia", [0,0,-3,-666])
pulp = ("Pulp Fiction", [9,10,9])
fc = ("Fight Club", [3,8,8,9,9,10])

pelis = [narnia,pulp,fc]
```

-- también tengo que definir funciones para interactuar con alumnos y películas

```
nombreAlumno unAlumno = fst unAlumno
notasAlumno unAlumno = snd unAlumno
tituloPelicula unaPelicula = fst unaPelicula
puntajesPelicula unaPelicula = snd unaPelicula
```

Si usamos lo que definimos arriba como un solo programa (un solo .hs), podemos ver que

1. La función `nombreAlumno` es igual a la función `tituloPelicula`
2. La función `notasAlumno` es igual a la función `puntajesPelicula`

Ejemplo

```
> fst fede
"Federico"

> snd fede
[2,3]

> fst fc
"Fight Club"

> snd narnia
[0,0,-3,-666]

> nombreAlumno fede
"Federico"

> nombreAlumno fc
"Fight Club"

> puntajesPelicula lider
[10,10,10,10,10]

> puntajesPelicula pulp
[9,10,9]

> empezaronMal cursoK9
[("Federico",[2,3])]

> empezaronMal pelis
[("Pedornia", [0,0,-3,-666]), ("Fight Club", [3,8,8,9,9,10])]
```

Todo esto es posible porque si miramos los tipos que infiere Haskell

```
fst :: (a,b) -> a
nombreAlumno :: (a,b) -> a
tituloPelicula :: (a,b) -> a
snd :: (a,b) -> b
notasAlumno :: (a,b) -> b
puntajesPelicula :: (a,b) -> b

fede :: (String, [Integer])
lider :: (String, [Integer])
ger :: (String, [Integer])

cursoK9 :: [ (String, [Integer]) ]

empezaronMal :: (Ord b) => [ (a,[b]) ] -> [ (a,[b]) ]

narnia :: (String, [Integer])
pulp :: (String, [Integer])
fc :: (String, [Integer])

pelis :: [ (String, [Integer]) ]
```

En ningún momento hablamos de Alumno o Pelicula, para Haskell los alumnos y peliculas son solo tuplas de 2 elementos

Ejemplo:

```
> nombreAlumno (True,"hola")
True

> puntajesPelicula ([1,2,3],(True,"hola"))
(True,"hola")
```

Ya que Haskell es un lenguaje "que se fija mucho en los tipos", nos gustaría que un caso como los de arriba nos tirará error (donde en vez de mandar un alumno o una película según corresponda, envíamos cualquier otra cosa).

Definiendo nuevos tipos

Para poder diferenciar a un alumno de una película y a ambos de una tupla, tenemos que definir un nuevo tipo.

Eso se hace usando **data**

```
data NuevoTipo = Constructor Tipo1 Tipo2 ... Tipon
```

En nuestro ejemplo

```
data TipoAlumno = Alumno String [Int]
data TipoPelicula = Pelicula String [Int]
```

-- Ahora, para obtener un nuevo alumno o una nueva película, tenemos que usar el "Constructor"

-- Usamos el constructor
```
fede = Alumno "Federico" [2,3]
lider = Alumno "Líder" [10,10,10,10,10]
ger = Alumno "Germain" [8,9,10]
```

-- No cambia
```
cursoK9 = [fede,lider,ger]
```

-- No cambia
```
empezaronMal unosAlumnos = filter ((4>).head.notasAlumno) unosAlumnos
```

-- Usamos el constructor
```
narnia = Pelicula "Pedornia" [0,0,-3,-666]
pulp = Pelicula "Pulp Fiction" [9,10,9]
fc = Pelicula "Fight Club" [8,8,8,9,9,10]
```

-- No cambia
```
pelis = [narnia,pulp,fc]
```

-- Ahora estas funciones usan Pattern-Matching!
```
nombreAlumno (Alumno nombre notas) = nombre
notasAlumno (Alumno nombre notas) = notas
```

```
tituloPelicula (Pelicula nombre notas) = nombre
puntajesPelicula (Pelicula nombre notas) = notas
```

Es importante remarcar que al hacer esto un alumno o una película **YA NO ES UNA TUPLA**

```
fst :: (a,b) -> a
nombreAlumno :: TipoAlumno -> String
tituloPelicula :: TipoPelicula -> String
snd :: (a,b) -> b
notasAlumno :: TipoAlumno -> [Int]
puntajesPelicula :: TipoPelicula -> [Int]

fede :: TipoAlumno
lider :: TipoAlumno
ger :: TipoAlumno

cursoK9 :: [ TipoAlumno ]

empezaronMal :: [ TipoAlumno ] -> [ TipoAlumno ]

narnia :: TipoPelicula
pulp :: TipoPelicula
fc :: TipoPelicula

pelis :: [ TipoPelicula ]
```

Ejemplos:

```
> fst fede
Error (fst espera una tupla y fede es de TipoAlumno)

> snd fede
Error (snd espera una tupla y fede es de TipoAlumno)

> fst fc
Error (fst espera una tupla y fc es de TipoPelicula)

> snd narnia
Error (snd espera una tupla y fc es de TipoPelicula)

> nombreAlumno fede
"Federico"

> nombreAlumno fc
Error (nombreAlumno espera un TipoAlumno y fc es de TipoPelicula)

> puntajesPelicula lider
Error (puntajesPelicula espera un TipoPelicula y lider es de TipoAlumno)

> puntajesPelicula pulp
[9,10,9]

> empezaronMal cursoK9
```

```
[Alumno "Federico" [2,3]]

> empezaronMal pelis
Error (empezaronMal espera una [TipoAlumno] y pelis es una [TipoPelicula])

> filter ((4>).head.puntajesPelicula) pelis
[Pelicula "Pedornia" [0,0,-3,-666], Pelicula "Fight Club" [3,8,8,9,9,10]]
```

Deriving

Es muy común querer comparar por igualdad y mostrar por pantalla a un valor que tiene un tipo definido por nosotros.

Ejemplo

```
> empezaronMal cursoK9
Error (el TipoAlumno no tiene la restricción Show)
```

Para que esto funcione deberíamos

1. Decir que `TipoAlumno` es un tipo que pertenece a la restricción `Show`
2. Definir la función `show` para un `TipoAlumno`

En vez de hacer esto a mano, y debido a que los campos que forman un `Alumno` **SI** tienen la restricción `Show`, podemos hacer que el `TipoAlumno` "derive" esa restricción

--Lo único que hay que agregar es **deriving (Show)**

```
data TipoAlumno = Alumno String [Int] deriving (Show)
```

Con este agregado podemos hacer

```
> empezaronMal cursoK9
[Alumno "Federico" [2,3]]
```

Ahora si hacemos lo siguiente

```
> fede == ger
Error (el TipoAlumno no tiene la restricción Eq)
```

También parece común querer preguntar si dos alumnos son iguales (o distintos), pasa lo mismo que con `Show`, nos gustaría que el `TipoAlumno` pertenezca a la restricción `Eq`

--Lo único que hay que agregar es **deriving (Show,Eq)**

```
data TipoAlumno = Alumno String [Int] deriving (Show,Eq)
```

Con este agregado podemos hacer

```
> fede == ger
False

> Alumno "Roberto" [7,8,9] == Alumno "Huberto" [7,8,9]
False

> Alumno "Roberto" [7,8,9] == Alumno "Roberto" [7,8,9]
True
```

OPCIONAL (BONUS 1): Agrandando nuestro sistema

Es muy común hacer funciones para obtener los valores que forman nuestro individuo compuesto.

Imaginen que ahora queremos agregarle a nuestro `TipoPelicula` (además del nombre y sus puntajes), el nombre del director, el nombre de los actores principales y el año en que se estreno.

```
data TipoPelicula = Pelicula String String [String] Int [Int]
```

Lo primero que notamos es que no es tan fácil identificar cada campo. Para eso existe la posibilidad de declarar **sinónimos de tipo** usando `type`.

Un sinónimo muy útil que **ya viene definido en Haskell** es

-- Se usa `type` cuando queremos declarar un sinónimo de tipos
```
type String = [Char]
```

No existe un tipo `String`, de hecho cuando preguntamos a Haskell siempre nos responde `[ Char ]`

```
> :t "hola"
"hola" :: [ Char ]
```

Pero podemos usar `String` en nuestras definiciones como sinónimo de `[ Char ]`

En el ejemplo de las películas podemos hacer algo como

```
type Titulo = String
type NombreDirector = String
type Puntajes = [Int]

data TipoPelicula = Pelicula Titulo NombreDirector [String] Int Puntajes
deriving (Show,Eq)
narnia = Pelicula "Pedornia" "Andrew Adamson" ["Tilda Swinton", "Georgie
Henley","William Moseley"] 2005 [0,0,-3,-666]
pulp = Pelicula "Pulp Fiction" "Quentin Tarantino" ["John Travolta", "Uma
Thurman", "Samuel L. Jackson"] 1994 [9,10,9]
fc = Pelicula "Fight Club" "David Fincher" ["Brad Pitt", "Edward Norton",
"Helena Bonham Carter"] 1999 [8,8,8,9,9,10]
```

Lo cual mejora la expresividad de la definición.

Otro tema es que tenemos que definir nuevamente funciones como `tituloPelicula` y `puntajesPelicula`:

```
tituloPelicula (Pelicula nombre director actores anioEstreno notas ) =
nombre
puntajesPelicula (Pelicula nombre director actores anioEstreno notas ) =
notas
```

Como en cualquier otro programa, las variables que no nos interesan en absoluto pueden ser reemplazadas por la variable anónima

```
tituloPelicula (Pelicula nombre _ _ _ _ ) = nombre
```

```
puntajesPelicula (Pelicula _ _ _ _ notas ) = notas
```

--También tenemos que definir funciones para el resto de los campos

```
directorPelicula (Pelicula _ director _ _ _ ) = director
actores (Pelicula _ _ actores _ _ ) = actores
anioEstreno (Pelicula _ _ _ anio _ ) = anio
```

Una forma más rápida de definir este tipo de funciones es usando **Type Record**s (solo disponible en GHC, no en Hugs)

En vez de definir solo los tipos de los valores que van a estar en la película, también agregamos en la definición el nombre de la función por el cual queremos obtener dicho valor

-- Al utilizar la notación de registro hay que encerrar la definición de los campos entre llaves { } y separar cada campo por comas ,

```
data TipoPelicula =
    Pelicula
        {tituloPelicula :: String ,
        directorPelicula :: String,
        actores :: [String],
        anioEstreno :: Int,
        puntajesPelicula :: [Int]}
        deriving (Show,Eq)
```

Con esta definición automaticamente Haskell define por nosotros las funciones `tituloPelicula`, `puntajesPelicula`, `directorPelicula`, `actores` y `anioEstreno`. El dominio de cada una de estas funciones es `TipoPelicula` y retornan lo que corresponda en cada caso.

Además cuando querramos obtener un nuevo Alumno, podemos hacer

```
narnia = Pelicula "Pedornia" "Andrew Adamson" ["Tilda Swinton", "Georgie
Henley","William Moseley"] 2005 [0,0,-3,-666]
pulp = Pelicula "Pulp Fiction" "Quentin Tarantino" ["John Travolta", "Uma
Thurman", "Samuel L. Jackson"] 1994 [9,10,9]
fc = Pelicula "Fight Club" "David Fincher" ["Brad Pitt", "Edward Norton",
"Helena Bonham Carter"] 1999 [8,8,8,9,9,10]
```

O bien

-- Usando la notación **record** es más claro a que campo pertenece cada valor

```
narnia =
    Pelicula{
        tituloPelicula = "Pedornia",
        directorPelicula = "Andrew Adamson",
        actores = ["Tilda Swinton", "Georgie Henley","William Moseley"],
        anioEstreno = 2005,
        puntajesPelicula = [0,0,-3,-666]}
```

-- Usando la notación **record** no es necesario seguir un orden en los valores mientras se indique a que campo pertenece

```
pulp =
```

```
    Pelicula{
        tituloPelicula = "Pulp Fiction",
        directorPelicula = "Quentin Tarantino",
        anioEstreno = 1994,
        puntajesPelicula = [9,10,9],
        actores = ["John Travolta", "Uma Thurman", "Samuel L. Jackson"]}

fc =
    Pelicula{
        directorPelicula = "David Fincher",
        anioEstreno = 1999,
        puntajesPelicula = [8,8,8,9,9,10],
        actores = ["Brad Pitt", "Edward Norton", "Helena Bonham Carter"],
        tituloPelicula = "Fight Club"}
```

Así como obtenemos todas estás ventajas, con la notación **record** tenemos la desventaja de que escribimos más. Uno tiene que evaluar cuando vale la pena y cuando no.

OPCIONAL (BONUS 2): Formalizando algunos temas

Ya dijimos que a cada restricción se la conoce como Clase/**Class**.

A cada tipo que pertenece a una Clase se lo conoce como Instancia/**Instance**.

Por ejemplo la clase Eq en algún lugar del Prelude (la biblioteca standard de Haskell) puede estar definida así

```
-- Esto ya viene con Haskell
class Eq a where
    (==), (/=) :: a -> a -> Bool
     -- Las instancias de Eq deben definir al menos una de estas 2 operaciones
    (/=) x y =  not (x == y)
    (==) x y =  not (x /= y)
```

Si decimos que el tipo `Bool` pertenece a la clase `Eq` escribimos

```
-- Esto ya viene con Haskell
instance Eq Bool where
    (==) True True = True
    (==) False False = True
    (==) _ _ = False
```

Otro ejemplo con la clase Ord

```
-- Notar que a tiene la restricción Eq en la definición de la clase Ord a
class Eq a => Ord a where
   (<) :: a -> a -> Bool
   (<=) :: a -> a -> Bool
   (>) :: a -> a -> Bool
   (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

Si queremos hacer que el `TipoPelicula` sea instancia de la clase `Ord`, podemos escribir

```
instance Ord TipoPelicula where
   -- Por poner un ejemplo, definimos la función (>) para que nos diga que una película es mayor que otra si su promedio de puntajes es mayor
   -- La función mayor que está en negrita > va a recibir por parámetro números por lo cual va a usar otra definición
     (>) unaPelicula otraPelicula = promedio (puntajesPelicula unaPelicula) > promedio (puntajesPelicula otraPelicula)
```