



Paradigma Funcional

Módulo 3: Modelado de información

**por Fernando Dodino
Carlos Lombardi
Nicolás Passerini
Daniel Solmirano
Revisión: Lucas Spigariol
Versión 2.1
Abril 2018**

Contenido

[1 Introducción](#)

[2 Listas](#)

[2.1 Definición](#)

[2.2 Formas de generar una lista](#)

[2.3 Pattern matching sobre listas](#)

[2.4 Ejercitando pattern matching sobre listas](#)

[2.5 Ejemplo didáctico](#)

[2.6 Funciones que aplican sobre listas](#)

[2.6.1 Head: cabeza de una lista](#)

[2.6.2 Tail: cola de una lista](#)

[2.6.3 Otras funciones](#)

[3 Tuplas](#)

[3.1 Definición](#)

[3.2 Pattern matching sobre tuplas](#)

[3.3 Ejemplos de tuplas](#)

[3.4 Comparación entre tuplas y listas](#)

[3.5 Funciones que aplican sobre tuplas](#)

[3.5.1 Sinónimos de tipo](#)

[3.5.2 Definición de una función con tuplas](#)

[3.6 El menor de un par](#)

[4 Tipos propios](#)

[4.1 Definición](#)

[4.2 Pattern matching sobre data](#)

[4.3 Diferencias entre data y tupla](#)

[4.4 Repaso de acoplamiento](#)

[4.5 Record syntax](#)

[4.6 Cómo definir una persona con record syntax](#)

[4.7 Funciones “adquiridas” con record syntax](#)

[4.8 Mostrar personas en la consola](#)

[5 Ejercicio integrador](#)

[5.1 Alumnos](#)

[5.2 Requerimientos](#)

[5.3 Modelar un parcial](#)

[5.4 Modelar el tipo del criterio de estudio](#)

[5.5 Modelar un alumno genérico](#)

[5.6 Representar los criterios de estudio enunciados](#)

[5.7 Modelar a Nico, un alumno estudiioso](#)

[5.8 Nico pasa de estudiioso a hijo del rigor](#)

[5.9 Saber si un alumno va a estudiar para el parcial de Paradigmas](#)

[6 Múltiples constructores](#)

[7 Resumen](#)

1 Introducción

Hasta el momento hemos conocido tipos de datos simples (los números, los valores booleanos, los caracteres), algunos compuestos (como las cadenas de caracteres o strings) y también las funciones como tipos de dato. Para modelar información, ampliaremos el universo conocido a tres tipos de dato compuestos:

- las listas
- las tuplas
- y los tipos de dato definidos por el usuario

2 Listas

2.1 Definición

Una lista es una serie de elementos del mismo tipo. En esa serie puede no haber elementos, en ese caso la lista es vacía. Son útiles para modelar las materias aprobadas por un alumno, las mascotas que tiene una persona, los empleados de una compañía, etc.

2.2 Formas de generar una lista

La forma más sencilla de generar una lista es enumerar los elementos encerrados entre corchetes y separar cada elemento mediante una coma

```
[1, 2, 3]  
["Hola", "mundo"]
```

Los strings son literales especiales que se encierran entre comillas dobles:
"Hola" ==> lista de caracteres ≡ ['H', 'o', 'l', 'a']

En una lista de Haskell todos los elementos deben ser homogéneos, de lo contrario se generará un mensaje de error:

```
ghc> [ "hola", True]  
Couldn't match expected type '[Char]' with actual type 'Bool'  
In the expression: True  
In the expression: ["hola", True]
```

En el caso de las listas numéricas, se puede generar una lista de enteros especificando cota inferior y superior:

```
ghc> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

También podemos definir una serie numérica de enteros a partir del primer número, el siguiente y la cota superior:

```
ghc> [1, 3..16]
[1,3,5,7,9,11,13,15]
```

El rango puede ser ascendente o descendente (dependiendo del segundo número y las cotas inferior o superior):

```
ghc> [8, 7.. -2]
[8,7,6,5,4,3,2,1,0,-1,-2]
```

E incluso podemos definir una lista infinita:

```
ghc> [1..]
[1,2,3,4,5,6,7..^C para cortar la evaluación
```

2.3 Pattern matching sobre listas

Existen varios patrones para trabajar las listas:

| Pattern | Denota |
|----------|--|
| [] | Lista vacía, no se puede separar en cabeza y cola |
| (x:xs) | El operador : separa cabeza y cola de una lista. Lista que tiene al menos un elemento, donde la cabeza es un elemento x, y la cola es una lista (son muchos x, por eso la convención es xs). Algunos ejemplos: <ul style="list-style-type: none"> • para [1, 2, 3] la cabeza es 1, la cola es la lista [2, 3]. • para [1] la cabeza es 1, la cola es [] (lista vacía) • para lista vacía, no hay pattern matching posible (una lista vacía no se puede separar en cabeza y cola) |
| (x:y:ys) | Lista de al menos 2 elementos. Algunos ejemplos: <ul style="list-style-type: none"> • [1, 2, 3], entonces x = 1, y = 2, ys = [3] • [1, 2, 3, 4], entonces x = 1, y = 2, ys = [3, 4] • [1, 2], x = 1, y = 2, ys = [] • [1], [] no producen pattern matching |
| [x] | Lista de exactamente un elemento. Algunos ejemplos: <ul style="list-style-type: none"> • para [1] x es 1 • para [1, 2] no hay pattern matching • tampoco hay pattern matching para lista vacía |
| [x, y] | Lista de exactamente dos elementos. Algunos ejemplos: |

- para `[1, 2]` $x = 1, y = 2$
- para `[]`, `[1]`, `[1, 2, 3]` no hay pattern matching

2.4 Ejercitando pattern matching sobre listas

Encontrar la cabeza y la cola de estas listas.

| Lista | Cabeza | Cola |
|--------------------------------------|---|---|
| <code>[1, 4]</code> | 1 | <code>[4]</code> ojo, no es 4, es la lista compuesta por 4 |
| <code>[]</code> | No existe, no se puede dividir | |
| <code>[[1, 7], [8, 7, 5], []]</code> | <code>[1, 7]</code> sí, es una lista | <code>[[8, 7, 5], []]</code> . Se puede usar listas de listas |
| <code>[8, "hermanos"]</code> | No es una lista posible en este lenguaje que define una lista como un tipo de dato recursivo, compuesto por elementos del mismo tipo. | |

2.5 Ejemplo didáctico

Para fijar el concepto llamamos a cuatro personas, cada una representa un valor distinto 1, 2, 3 y 4:



¿Qué lista se forma? `[1, 2, 3, 4]`,
¿cuál es la cabeza? 1, ¿cuál es la cola? `[2, 3, 4]` (que se lee "la lista compuesta por 2, 3 y 4")

Bien, el que estaba en la cabeza se va a sentar y seguimos con la lista resultante. ¿Cuál es la cabeza y cuál es la cola? 2, y `[3, 4]`. Lo importante es recalcar que la cola **siempre** es una lista: `[3,4]` se divide en cabeza 3 y cola `[4]` ("la lista compuesta por 4"). Y `[4]` es una lista cuya cabeza es 4 y la cola es la lista vacía.

Entonces, la lista con los números `[1, 2, 3]` puede escribirse:

`(1:[2, 3]) = (1:(2:[3])) = (1:(2:(3:[])))`

2.6 Funciones que aplican sobre listas

2.6.1 Head: cabeza de una lista

La función head/1 devuelve la cabeza de una lista.

```
ghc> head [1, 7, 9]
1
```

Implementamos la función aplicando pattern matching:

```
head (x:xs) = x
```

Como xs no nos interesa utilizarlo en la función, podemos utilizar la variable anónima (que se denota con el símbolo guión bajo o *underscore*):

```
head (x:_ ) = x
```

¿Qué tipo de lista podemos recibir? Algunas pruebas en consola:

```
ghc> head ["hola" , "mundo"]  
"hola"
```

```
ghc> head [[], [1,2], [3, 4]]  
[]
```

Podemos recibir listas de cualquier tipo:

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

y devolverá la cabeza de esa lista, que será del mismo tipo que el resto de los elementos.

2.6.2 Tail: cola de una lista

De la misma manera tenemos la función que devuelve la cola de una lista:

```
tail (_:xs) = xs
```

¿De qué tipo es la función tail?

```
tail :: [a] -> [a]
```

2.6.3 Otras funciones

Existen otras funciones que desarrollaremos próximamente, pero que ya podemos usar¹:

| Función | Objetivo | Ejemplo de uso |
|---------|--|--------------------------------------|
| length | Devuelve la longitud de una lista | ghc> length [2..4] 3 |
| sum | Suma los elementos de una lista de números | ghc> sum [4, 9, 1] 14 |
| (++) | Concatena dos listas | ghc> [3..5] ++ [4, 6] [3,4,5,4,6] |
| take | Toma los primeros n elementos de la lista | ghc> take 3 [4..11] [4,5,6] |

¹ Para más información consultar la [guía de lenguajes](#)

| | | |
|---------|--|--|
| drop | Devuelve la lista sin los primeros n elementos | <code>ghc> drop 2 "CIENCIA" "ENCIA"</code> |
| (!!) | Devuelve el elemento que está en la posición n (donde 0 es el primer elemento) | <code>ghc> [1..10] !! 3 4 ghc> ["buen", "dia", "gente", "como", "andan"] !! 2 "gente"</code> |
| reverse | Devuelve una lista con los elementos en orden inverso | <code>ghc> reverse "ANITA LAVA LA TINA" "ANIT AL AVAL ATINA"</code> |

3 Tuplas

3.1 Definición

Las tuplas permiten representar un tipo de dato compuesto, pero con elementos que pueden ser de distinto tipo. El número de elementos es fijo (siempre el mismo).

(23, 02, 1973) => tupla de 3 elementos para representar una fecha
("Juan", 23) => tupla de 2 elementos para representar una persona

Una tupla es un tipo de dato compuesto donde
(x₁, x₂.. x_n) tiene tipos t₁, t₂... t_n

3.2 Pattern matching sobre tuplas

No son muchos los patrones que describen tuplas

| Pattern | Denota |
|-----------|---|
| (x, y) | Tupla de dos elementos, x e y pueden tomar valores de cualquier tipo |
| (x, y, z) | Tupla de tres elementos, x, y, z pueden tomar valores de cualquier tipo |

3.3 Ejemplos de tuplas

Algunos ejemplos de tuplas: ¿De qué tipo son?

| Tupla | Tipo de la tupla |
|--------------------------|--------------------------|
| (1, [2], "ultravioleta") | (Int, [Int], String) |
| ('a', False) | (Char, Bool) |
| ((1,2),(3,4)) | ((Int, Int), (Int, Int)) |

3.4 Comparación entre tuplas y listas

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

3.5 Funciones que aplican sobre tuplas

Hay funciones ya definidas para separar una tupla de dos elementos: *fst* devuelve el primer elemento de la tupla, *snd* devuelve el segundo elemento.

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

¿De qué tipo son?

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

Veamos un ejemplo de uso en la consola:

```
ghc> snd ("Laura", True)
True
ghc> snd ("Laura", (30, 6))
(30,6)
```

En el segundo ejemplo, vemos que la tupla puede contener como elemento otra tupla (que represente, por ejemplo, la fecha de cumpleaños). También podríamos usar una lista, para representar un alumno con sus notas:

```
tobi = ("Tobias Sandler", [4, 8, 9])
```

Pero en lugar de utilizar *snd*, podemos definir una función *notas*, que es mucho más representativo del dominio que estamos modelando²:

```
notas (_, n) = n
```

² Lo que en alguna bibliografía se conoce como *intention revealing*

o aun mejor, utilizando la igualdad en el sentido matemático:

`notas = snd`

```
ghc> notas tobi  
[4, 8, 9]
```

Esto es algo a tener en cuenta a la hora de elegir una tupla como forma de representar información de nuestro negocio.

3.5.1 Sinónimos de tipo

El requerimiento que nos piden es “modelar la suma entre números complejos”. La primera decisión que tomamos es que usaremos una tupla para representar un número complejo.

Definimos entonces un sinónimo de tipo, para decir que existen los números complejos, de la siguiente manera:

```
type Complejo = (Float, Float)
```

3.5.2 Definición de una función con tuplas

Escribimos el tipo que tiene la función que suma complejos:

```
sumarComplejos :: Complejo -> Complejo -> Complejo
```

Codificamos la función:

```
sumarComplejos (real1, imaginario1) (real2, imaginario2) =  
    (real1 + real2, imaginario1 + imaginario2)
```

Y por último lo probamos en la consola:

```
ghc> sumarComplejos (1, 2) (4, -1)  
(5.0, 1.0)
```

3.6 El menor de un par

Otra función que calcule el menor de un par de números puede ser escrita así:

```
minPar (numero1, numero2) = min numero1 numero2
```

4 Tipos propios

4.1 Definición

Se pueden definir nuevos tipos de datos que se agregan a los ya existentes en el lenguaje, mediante la expresión `data`. Si queremos modelar varias personas que tienen nombre y edad, lo hacemos de la siguiente manera:

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

Esto se lee como: "el tipo de dato Persona (en azul) utiliza un constructor -de nombre Persona, en rojo- que recibe un String y un Int".

4.2 Pattern matching sobre data

Para poder reconocer un tipo de dato propio usamos el constructor: en el caso **Persona** "Santiago" **32**, el primer parámetro (String que representa el nombre) se asocia a "Santiago" y el segundo (Int que representa la edad) a **32**

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

```
nombre (Persona _nombre _edad) = _nombre
edad (Persona _nombre _edad) = _edad
```

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:

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

4.3 Diferencias entre data y tupla

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

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

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

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

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:

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

`Persona "Miguel" 34 :: Persona`

De hecho la aplicación parcial funciona perfectamente:

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

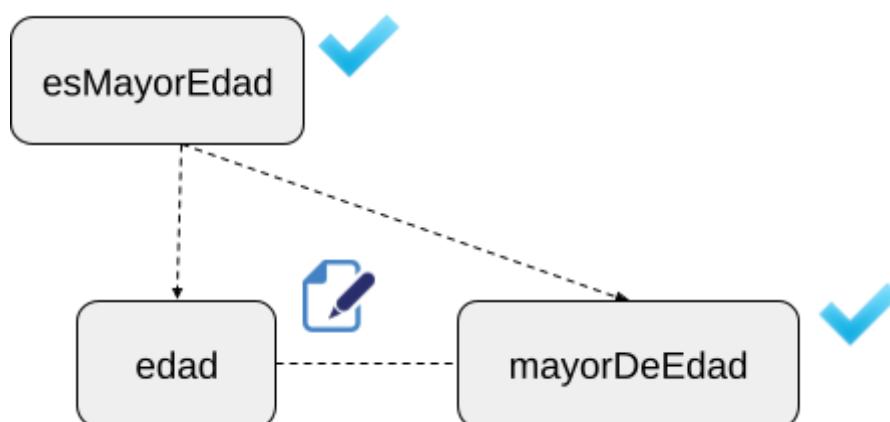
Es decir, Persona "Miguel" es una función que espera la edad y devuelve una persona.

4.4 Repaso de acoplamiento

Recordemos la función que resuelve el requerimiento "Saber si una persona es mayor de edad":

```
esMayorEdad :: Persona -> Bool
esMayorEdad = mayorDeEdad . edad
```

Con la nueva definición de Persona como registro, no tenemos cambios en la función esMayorEdad



lo cual sigue demostrando que hay un buen grado de acoplamiento entre

- las funciones `esMayorEdad` y `edad`
- la función `esMayorEdad` y la definición de `Persona` (antes tupla, hoy data)

4.5 Record syntax

Se agrega información adicional a la persona:

- el domicilio, que vamos a modelar con un String
- el número de teléfono, un nuevo String
- la fecha de nacimiento, que puede ser una tupla de tres elementos
- si es buena persona, un booleano

- y la cantidad de plata que tiene en el bolsillo, un número con decimales

El constructor Persona se vuelve bastante menos expresivo:

```
data Persona = Persona String Int String String (Int, Int, Int) Bool
Float
```

sin contar que además necesitaremos funciones que se incorporan a nombre y edad...

```
domicilio :: Persona -> String
domicilio (Persona _ _ dom _ _ _ _ ) = dom
```

... y lo mismo con cada uno de los datos que componen la estructura. Para sumar expresividad y evitar definiciones que podríamos considerar “redundantes”, tenemos la notación *record syntax* que consiste en definir Persona de la siguiente manera:

```
data Persona = Persona {
    nombre :: String,
    edad :: Int,
    domicilio :: String,
    telefono :: String,
    fechaNacimiento :: (Int, Int, Int),
    buenaPersona :: Bool,
    plata :: Float
}
```

4.6 Cómo definir una persona con record syntax

Para crear una persona podemos utilizar el mismo formato que utilizamos anteriormente

```
juan = Persona "Juan" 29 "Ayacucho 554" "45232598" (17,7,1988) True
30.0
```

Esto como vemos

- es una definición propensa a cometer errores
- tiene poca expresividad, no siempre los valores revelan de qué información se trata
- nos obliga a cargar la información respetando el orden que definió el tipo de dato

Pero como el tipo de dato Persona está definido con *record syntax*, tenemos una alternativa para modelar a Juan:

```
juan = Persona {  
    nombre = "Juan",  
    edad = 29,  
    domicilio = "Ayacucho 554",  
    telefono = "45232598",  
    fechaNacimiento = (17,7,1988),  
    buenaPersona = True,  
    plata = 30.0  
}
```

No solo es más expresiva la forma de definir la información, sino que también podemos alterar el orden en el que definimos los valores de juan:

```
juan = Persona {  
    nombre = "Juan",  
    telefono = "45232598",  
    domicilio = "Ayacucho 554",  
    fechaNacimiento = (17,7,1988),  
    buenaPersona = True,  
    edad = 29,  
    plata = 30.0  
}
```

4.7 Funciones “adquiridas” con record syntax

Al definir una Persona con la sintaxis de record, adquirimos funciones que permiten preguntar por las propiedades de una persona:

```
ghc> telefono juan  
"45232598"  
ghc> domicilio juan  
"Ayacucho 554"
```

No necesitamos (*ni podemos*) escribir definiciones de la función telefono, edad ni domicilio. Éstas están implícitas por la definición del tipo de dato Persona:

```
ghc> :t edad  
edad :: Persona -> Int
```

4.8 Mostrar personas en la consola

Si intentamos visualizar a Juan en la consola, aparece un mensaje de error:

```
ghc> juan
<interactive>:14:1:
    No instance for (Show Persona) arising from a use of `print'
    In a stmt of an interactive GHCi command: print it
```

Si bien más adelante estaremos estudiando en detalle el sistema de tipos de Haskell, podemos explicar brevemente que una Persona no se puede mostrar si no se define como instancia de Show. Para solucionar este error, basta con incorporar este agregado a nuestra definición de Persona:

```
data Persona = Persona {
    nombre :: String,
    edad :: Int,
    domicilio :: String,
    telefono :: String,
    fechaNacimiento :: (Int, Int, Int),
    buenaPersona :: Bool,
    plata :: Float
} deriving (Show)
```

Ahora sí, podemos evaluar en la consola a Juan:

```
ghc> juan
Persona {nombre = "Juan", edad = 29, domicilio = "Ayacucho 554",
telefono = "45232598", fechaNacimiento = (17,7,1988), buenaPersona =
True, plata = 30.0}
```

5 Ejercicio integrador

Hasta aquí ya conocemos una amplia variedad de valores que nos pueden ayudar a modelar información

- **simples:** números, booleanos, caracteres... y no nos olvidemos de las *funciones* que también son valores posibles
- **compuestos:** strings, listas, tuplas, data

5.1 Alumnos

Modelar un alumno, que define

- un nombre,
- la fecha de nacimiento,

- el legajo (sin dígito verificador),
- las materias que cursa
- y el criterio para estudiar ante un parcial:
 - algunos son estudiados: estudian siempre,
 - otros son hijos del rigor: estudian si el parcial tiene más de n preguntas,
 - y también están los cabuleros, que estudian si la materia tiene una cantidad impar de letras.

5.2 Requerimientos

1. Modelar un parcial
2. Modelar el tipo que representa el criterio de estudio.
3. Modelar genéricamente un alumno.
4. Representar con la abstracción que crea más conveniente al criterio estudiado, hijo del rigor y cabulero.
5. Modelar a Nico, un alumno estudiado
6. Hacer que Nico pase de ser estudiado a hijo del rigor (buscar una abstracción lo suficientemente genérica)
7. Determinar si Nico va a estudiar para el parcial de Paradigmas

5.3 Modelar un parcial

Para resolver este requerimiento, tenemos que abstraer la información necesaria para poder resolver lo que hoy nos piden. Hay mucha información que un parcial puede tener:

- hora de comienzo
- hora de fin
- el/la docente que lo toma
- la materia
- el aula
- la cantidad de alumnos presentes
- la cantidad de preguntas
- las preguntas propiamente dichas...

... entre otros datos. Pero en este enunciado, solamente nos importan dos cosas: la materia (para los cabuleros) y la cantidad de preguntas que tiene un parcial (para los hijos del rigor).

Esta técnica que empleamos toma en cuenta lo que nos interesa y descarta lo que no es esencial para la solución, proceso que se llama **abstracción**.

- ¿Cómo modelamos la materia? No necesitamos que sea una estructura compuesta, nos alcanza con que sea un String.
- Respecto a las preguntas, tampoco necesitamos conocer la lista de preguntas concreta, nos basta con saber la cantidad por el momento, entonces un entero es suficiente.

Describimos un Parcial como:

```
data Parcial = Parcial String Int deriving (Show)
```

¿Podría ser el parcial una tupla? Sí, y eso nos hubiera permitido usar fst / snd para obtener la materia / la cantidad de preguntas. Pero hemos visto que no es lo suficientemente expresiva esa solución, así que construiremos nuestras propias funciones (o bien podemos escribir al Parcial con *record syntax*):

```
materia :: Parcial -> String
materia (Parcial mat _) = mat

cantidadPreguntas :: Parcial -> Int
cantidadPreguntas (Parcial _ cant) = cant
```

5.4 Modelar el tipo del criterio de estudio

¿Qué representa el criterio? Dado un parcial, queremos saber si va a estudiar. Es decir:

```
Parcial -> Bool
```

Exacto, el criterio se modela con una función, que va a formar parte de la estructura de un alumno. Por eso no queremos usar:

- códigos numéricos (1 = estudia siempre, 2 = hijo del rigor, 3 = cabulero)
- ni códigos alfabéticos ("ES", "HR", "CA")
- ni ningún otro mecanismo de indirección: en la estructura del alumno debe estar el criterio que nos diga si va a estudiar ante un parcial.

Para definir el tipo, escribimos

```
type CriterioEstudio = Parcial -> Bool
```

5.5 Modelar un alumno genérico

Un alumno se puede representar como una estructura

- con tuplas
- o bien con un tipo de dato propio (*data*)

Dada la gran cantidad de información que se asocia a un alumno, vamos a elegir definir un tipo de dato propio con notación de registro (*record syntax*).

La siguiente decisión es analizar de qué tipo es cada uno de los datos del alumno:

- el nombre se modela con un String
- la fecha de nacimiento, con una tupla (aunque podría ser un data también)
- el legajo sin el dígito verificador se puede modelar con un Int
- las materias que cursa: no necesitamos representar a la materia como una entidad compuesta, así que con una lista de strings nos alcanza
- el criterio de estudio ya está modelado como una función

Escribimos:

```
data Alumno = Alumno {  
    nombre :: String,  
    fechaNacimiento :: (Int, Int, Int),  
    legajo :: Int,  
    materiasQueCursa :: [String],  
    criterioEstudio :: CriterioEstudio  
} deriving (Show)
```

5.6 Representar los criterios de estudio enunciados

Definimos entonces tres funciones:

El estudiioso que siempre estudia (no importa el parcial):

```
estudiioso :: CriterioEstudio  
estudiioso _ = True
```

El hijo del rigor, al que le podemos configurar la cantidad de preguntas por sobre las cuales comienza a estudiar:

```
hijoDelRigor :: Int -> CriterioEstudio  
hijoDelRigor n (Parcial _ preguntas) = preguntas > n
```

Y el cabulero que depende de que la materia tenga una cantidad impar de letras:

```
cabulero :: CriterioEstudio  
cabulero (Parcial materia _) = (odd . length) materia3
```

³ Pregunta para el lector: ¿se puede usar notación point free aquí? ¿por qué (no)?

5.7 Modelar a Nico, un alumno estudiioso

Aprovechamos la definición anterior de estudioso para escribir la expresión que describe a Nico:

```
nico = Alumno {  
    fechaNacimiento = (10, 3, 1993),  
    nombre = "Nico",  
    materiasQueCursa = ["sysop", "proyecto"],  
    criterioEstudio = estudioso,  
    legajo = 124124  
}
```

Evaluamos nico por consola y nos aparece un error:

```
No instance for (Show CriterioEstudio)  
  arising from the fifth field of `Alumno' (type  
`CriterioEstudio')  
  Possible fix:  
    use a standalone 'deriving instance' declaration,  
    so you can specify the instance context yourself  
When deriving the instance for (Show Alumno)
```

Al igual que nos pasaba con la persona, necesitamos decirle cómo mostrar las funciones en la consola. La solución, incorporamos esta línea al comienzo de nuestro .hs:

```
import Text.Show.Functions
```

Y ahora sí podemos ver representado a nico en consola:

```
ghc> nico  
Alumno {  
    nombre = "Nico",  
    fechaNacimiento = (10,3,1993),  
    legajo = 124124,  
    materiasQueCursa = ["sysop", "proyecto"],  
    criterioEstudio = <function>  
}
```

5.8 Nico pasa de estudioso a hijo del rigor

Como dentro del paradigma funcional queremos trabajar sin efecto colateral, no podemos modificar directamente a Nico, sino que **tenemos que generar una**

transformación que tome a un estudiante y devuelva un nuevo estudiante modificado.

Por eso generaremos una función cambiarCriterioEstudio:



Una primera forma de resolverlo puede ser:

```
cambiarCriterioEstudio nuevoCriterio alumno = Alumno {  
    nombre = nombre alumno,  
    fechaNacimiento = fechaNacimiento alumno,  
    legajo = legajo alumno,  
    materiasQueCursa = materiasQueCursa alumno,  
    criterioEstudio = nuevoCriterio  
}
```

Lo que resulta un tanto tedioso. Otra alternativa podría ser crear el nuevo alumno respetando el orden de la información definida en el tipo de dato:

```
cambiarCriterioEstudio nuevoCriterio alumno = Alumno  
(nombre alumno) (fechaNacimiento alumno)  
(legajo alumno) (materiasQueCursa alumno) nuevoCriterio
```

O con pattern matching

```
cambiarCriterioEstudio nuevoCriterio  
    (Alumno nombre fecha legajo materias _)  
    = Alumno nombre fecha legajo materias nuevoCriterio)
```

Esta opción permite generar un nuevo alumno indicando sólo lo que cambia⁴:

```
cambiarCriterioEstudio nuevoCriterio alumno = alumno {  
    criterioEstudio = nuevoCriterio  
}
```

⁴ Agradecemos por este tip a Juan Ignacio Ferro, del curso de verano 2017. Atención: esta variante es válida sólo si se trabaja con *record syntax*.

En la parte derecha de la función no estamos usando el constructor Alumno, sino la variable alumno que recibimos como parámetro.

¿De qué tipo es cambiarCriterioEstudio?

```
cambiarCriterioEstudio :: CriterioEstudio -> Alumno -> Alumno
```

Para pasar a Nico a ser hijo del rigor, necesitamos definir el mínimo de preguntas, de lo contrario no va a pasar el chequeo de tipos que hace el compilador Haskell:

```
ghc> cambiarCriterioEstudio hijoDelRigor nico
Couldn't match type `Int' with `Parcial'
Expected type: CriterioEstudio
    Actual type: Int -> CriterioEstudio
Probable cause: `hijoDelRigor' is applied to too few arguments
In the first argument of `cambiarCriterioEstudio', namely
`hijoDelRigor'
In the second argument of `($)', namely
`cambiarCriterioEstudio hijoDelRigor nico'
```

Exacto, hijoDelRigor es una función que va de Int a CriterioEstudio, es decir, de Int a Parcial -> Bool. Corregimos entonces esto utilizando aplicación parcial:

```
ghc> cambiarCriterioEstudio (hijoDelRigor 5) nico
Alumno {
    nombre = "Nico",
    fechaNacimiento = (10,3,1993),
    legajo = 124124,
    materiasQueCursa = ["sysop", "proyecto"],
    criterioEstudio = <function>
}
```

Todavía no se aprecia que devuelve una función diferente.

5.9 Saber si un alumno va a estudiar para el parcial de Paradigmas

Ok, necesitamos

- un alumno: podemos usar Nico
- un parcial de Paradigmas

Si estudia o no dependerá del criterio del alumno. Entonces:

```
estudia :: Parcial -> Alumno -> Bool
estudia parcial alumno = (criterioEstudio alumno) parcial
```

Podemos transformar a Nico de estudiioso a hijo del rigor con 5 preguntas y luego preguntar si estudia para el parcial de PDP que tiene 3 preguntas:

```
parcialPDP = Parcial "PDP" 3

ghc> (estudia parcialPDP . cambiarCriterioEstudio (hijoDelRigor 5))
nico
False

ghc> (estudia parcialPDP . cambiarCriterioEstudio (hijoDelRigor 2))
nico
True
```

Para el lector curioso, este mismo ejemplo está implementado en C y pueden descargarlo en <https://github.com/uqbar-project/eg-alumnos-c>

6 Múltiples constructores

Es posible definir un tipo de dato propio que admita más de un constructor. Por ejemplo podemos definir

```
data ColorPrimario = Rojo | Amarillo | Azul
```

Esto se lee: “el tipo de dato ColorPrimario tiene tres constructores: Rojo, Amarillo y Azul, todos sin parámetros”. En este caso sirve para representar una enumeración. Entonces podemos construir una función que discrimine ambos elementos por pattern matching

```
recargoPorColor :: ColorPrimario -> Int
recargoPorColor Rojo  = 50
recargoPorColor _     = 20 // por descarte, amarillo o azul
```

7 Resumen

A lo largo de este extenso capítulo hemos incorporado a las listas, las tuplas y los data como formas de modelar información compuesta, que se suman a los números, los caracteres, los booleanos, los strings y las funciones. Entonces, podemos combinar las abstracciones en una estructura que representa una

entidad, y operaciones asociadas a dicha estructura que pueden tener o no efecto colateral.