Pattern matching + Tipos enumerados

Taller de Álgebra I

Segundo cuatrimestre de 2016

Pattern matching

El **pattern matching** es un mecanismo que nos permite asociar una definición de una función solo a ciertos valores de sus parámetros: aquellos que se correspondan con cierto **patrón**.

Pattern matching en `Bool`

Si quisiéramos definir la función `not` (negación lógica), podríamos hacerlo así:

```
not :: Bool -> Bool
not x | x == True  = False
      | x == False = True
```

Acá, `x` es un **patrón**: es el menos restrictivo posible, ya que se corresponde con cualquier valor de tipo `Bool`.

El tipo `Bool` admite otros dos patrones más restrictivos: `True` y `False`. Usando estos patrones, podemos redefinir `not` de esta forma:

```
not :: Bool -> Bool
not True  = False
not False = True
```

Pattern matching en `Integer`

En el tipo `Integer`, todos los números son patrones válidos. Por ejemplo, podemos reescribir la función `factorial :: Integer -> Integer`

```
factorial n | n == 0 = 1
            | otherwise = n * factorial (n - 1)
```

usando *pattern matching*:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Para **reducir** cualquier expresión que contenga `factorial`, Haskell compara, en orden de arriba hacia abajo, cada patrón con los valores de los parámetros, y utiliza el primero que funcione.

Si el patrón tiene **variables libres**, se **ligan** a los valores de los parámetros.

Todos los tipos de datos admiten el patrón _, que se corresponde con cualquier valor, pero no liga ninguna variable. Lo usamos cuando no nos importa el valor de algún parámetro. Por ejemplo:

```
esLaRespuestaATodo :: Integer -> Bool
esLaRespuestaATodo 42 = True
esLaRespuestaATodo _  = False
```

Pattern matching en tuplas

El *pattern matching* también nos permite escribir de forma más clara definiciones que involucren **tuplas**.

Podemos usar patrones para descomponer la estructura de una tupla en los elementos que la forman y ligar cada uno de ellos a una variable distinta.

Por ejemplo, la siguiente definición:

```
sumaVectorial :: (Float, Float) -> (Float, Float) -> (Float, Float)
sumaVectorial t1 t2 = (fst t1 + fst t2, snd t1 + snd t2)
```

puede reescribirse como:

```
sumaVectorial :: (Float, Float) -> (Float, Float) -> (Float, Float)
sumaVectorial (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

En este caso, el patrón `(x1, y1)` se corresponde con la primera tupla, y las variables `x1` e `y1` se ligan con cada una de las componentes de la tupla. Algo análogo pasa con la segunda tupla y el patrón `(x2, y2)`.

Ejercicios

- ¿Son correctas las siguientes definiciones? ¿Por qué?

```
factorial 0       = 1
factorial (n + 1) = (n + 1) * factorial n
```

```
iguales :: Integer -> Integer -> Bool
iguales x x = True
iguales x y = False
```

- Escribir las definiciones de las siguientes funciones, **utilizando pattern matching**. Tratar de evaluar la mínima cantidad de parámetros necesaria.
  - `yLogico :: Bool -> Bool -> Bool`, la conjunción lógica.
  - `oLogico :: Bool -> Bool -> Bool`, la disyunción lógica.
  - `implica :: Bool -> Bool -> Bool`, la implicación lógica.
  - `sumaGaussiana :: Integer -> Integer`,
    que toma un entero no negativo y devuelve la suma de todos los enteros positivos menores o iguales que él.
  - `algunoEsCero :: (Integer, Integer, Integer) -> Bool`,
    que devuelve True sii alguna de las componentes de la tupla es 0.
  - `productoInterno :: (Float, Float) -> (Float, Float) -> Float`,
    que dados dos vectores $v_1 = (x_1, y_1)$, $v_2 = (x_2, y_2) \in \mathbb{R}^2$, calcula su producto interno $\langle v_1, v_2 \rangle = x_1 x_2 + y_1 y_2$.

Tipos enumerados

Un **tipo enumerado** es un tipo de datos cuyos valores posibles están dados *por extensión*.

El tipo `Bool` es un ejemplo de tipo enumerado; podemos definirlo así:

```
data Bool = False | True
```

- El término `data` indica que estamos definiendo un tipo de datos.
- `Bool` es el nombre del nuevo tipo de datos.
- A la derecha del = tenemos todos los **constructores**. Cada uno de ellos es un valor posible del nuevo tipo. Podemos pensarlos como funciones que *no reciben parámetros* y devuelven una instancia del tipo que estamos definiendo.
- La barra vertical (|) separa los distintos constructores.

Cada uno de los constructores constituye un patrón válido a la hora de hacer *pattern matching*.

Ejercicio: El tipo `Dia`

Ejercicios

- Definir un tipo de datos enumerado, `Dia`, cuyos valores posibles sean los días de la semana: {Domingo, Lunes, Martes, Miércoles, Jueves, Viernes, Sábado}.
- Definir las siguientes funciones:
  - `esFinde :: Dia -> Bool`,
    que determina si el día es parte del fin de semana.
  - `esDiaHabil :: Dia -> Bool`,
    que determina si el día es un día hábil.

¿Qué pasa cuando queremos probar estas funciones?

Repaso: Clases de tipos

Haciendo memoria

Las **clases de tipos** son *conjuntos de tipos de datos* que tienen definidas determinadas operaciones.

Algunas clases de tipos útiles:

- `Eq` es la clase de los tipos cuyos elementos pueden **compararse por igualdad**; tienen definidas las funciones `(==)` y `(\=)`, que devuelven un `Bool`.
- `Ord` es la clase de los tipos cuyos elementos pueden **ordenarse**; tienen definidas funciones como `(<=)`, `(>=)`, `(<)` y `(>)`, que devuelven un `Bool`.
- `Show` es la clase de los tipos cuyos elementos pueden **mostrarse por pantalla**; tienen definida la función `show`, que devuelve un `String`.

Nuestros tipos y las clases de tipos

Cuando definimos un tipo de datos nuevo, no forma parte de **ninguna clase**.

¡Esto quiere decir que Haskell no sabe cómo mostrar un elemento de nuestro tipo por pantalla! Tampoco puede, por ejemplo, saber si dos elementos son o no iguales.

Para que nuestro tipo sea parte de una clase, tenemos que *definir* las funciones necesarias. Por ejemplo:

- Para que un tipo `t` sea parte de `Eq`, hay que definir una función `(==) :: t -> t -> Bool`.
- Para que un tipo `t` sea parte de `Ord`, hay que definir una función `(<=) :: t -> t -> Bool`.
- Para que un tipo `t` sea parte de `Show`, hay que definir una función `show :: t -> String`.

La forma más sencilla de hacer esto es dejar que Haskell **infiera** la definición de estas funciones. Para esto, agregamos a la definición del tipo la palabra clave `deriving`, seguida de las clases que queramos.

Nuestros tipos y las clases de tipos

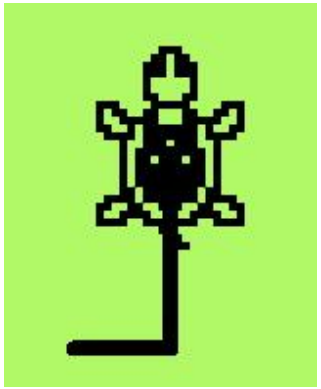
Ejercicios

- Hacer que `Dia` sea parte de las clases `Eq`, `Ord` y `Show`.

```
data Dia = Domingo | ... | Sabado deriving (Eq, Ord, Show)
```

- ¿Cómo define Haskell, por defecto, las funciones `(==)`, `(<=)` y `show`?
- Ahora sí, probar que las funciones `esFinde` y `esDiaHabil` funcionan correctamente.

Ejercicio: Logo



Teniendo en cuenta las siguientes definiciones:

```
type Posicion = (Integer, Integer)
data Direccion = Norte | Sur | Este | Oeste
type Tortuga = (Posicion, Direccion)
```

programar estas funciones:

1. `arrancar :: Tortuga`,
   que representa una tortuga en el `(0,0)` mirando hacia el `Sur`.
2. `girarDerecha :: Tortuga -> Tortuga`,
   que gira la tortuga 90 grados a la derecha, sin moverla de lugar.
3. `avanzar :: Tortuga -> Integer -> Tortuga`,
   que hace avanzar a la tortuga la distancia indicada, en la dirección hacia donde está mirando.