Pattern matching + Más recursión

Taller de Álgebra I

Verano 2018

Pattern matching

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**.

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`.

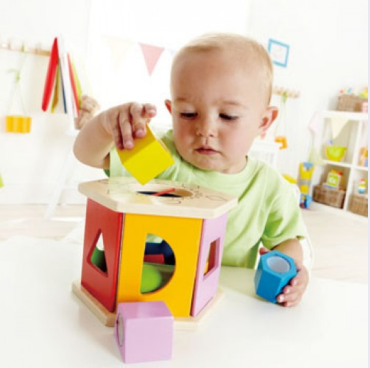
El tipo `Bool` admite otros dos patrones más restrictivos: `True` y `False`. Usando estos patrones, podemos redefinir `not` de esta forma:

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

Pattern matching: explicación gráfica

La siguiente función toma un polígono y nos dice cuántos lados tiene:

La función que nos dice la cantidad de lados



`cantidadLados`  `= 3`

`cantidadLados`  `= 4`

`cantidadLados`  `= 5`

`cantidadLados`  `= 6`

`cantidadLados`  `= 7`

`cantidadLados`  `= 8`

`cantidadLados`  `= 9`

`cantidadLados`  `= 10`

$\vdots$

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

Ejercicios

1. ¿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
```

2. Escribir las definiciones de las siguientes funciones, **utilizando pattern matching**. Tratar de evaluar la mínima cantidad de parámetros necesaria.

   (a) `yLogico :: Bool -> Bool -> Bool`, la conjunción lógica.

   (b) `oLogico :: Bool -> Bool -> Bool`, la disyunción lógica.

   (c) `implica :: Bool -> Bool -> Bool`, la implicación lógica.

   (d) `sumaGaussiana :: Integer -> Integer`,
   que toma un entero no negativo y devuelve la suma de todos los enteros positivos menores o iguales que él.

   (e) `algunoEsCero :: (Integer, Integer, Integer) -> Bool`,
   que devuelve True sii alguna de las componentes de la tupla es 0.

   (f) `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 $\langle v_1, v_2 \rangle = x_1 x_2 + y_1 y_2$.

Más recursión

Ejercicios (más recursión)

1. Implementar una función que dado un número natural n, determine si puede escribirse como suma de dos números primos: `esSumaDeDosPrimos :: Integer -> Bool`
2. **Conjetura (Christian Goldbach, 1742):** todo número par mayor que 2 puede escribirse como suma de dos números primos. Escribir una función que pruebe la conjetura hasta un cierto punto. `goldbach :: Integer -> Bool` (hasta al menos 4.10^{18} debería ser cierto)
3. Escribir una función que determine la suma de digitos de un número positivo. Para esta función pueden utilizar `div` y `mod`.
4. Implementar una función que determine si todos los dígitos de un número son iguales.
5. Sea la siguiente definición:

$$a_{n+1} = \begin{cases} \frac{a_n}{2} & \text{si } a_n \text{ es par} \\ 3a_n + 1 & \text{si } a_n \text{ es impar} \end{cases}$$

   Si comenzamos con $a_1 = 13$, obtenemos la siguiente secuencia: $13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$ (10 términos). No fue demostrado aun (problema de Collatz), que empezando con cualquier número siempre se termina en 1.

   - Resolver usando Haskell: ¿Qué número debajo de un 10.000 produce la secuencia más larga?