Haskell for OOP(rogrammers)

Programación Avanzada - UNLaM

24 de Junio de 2019


Haskell

¿Qué es Haskell?

1. Es un lenguaje de programación
2. Utiliza el paradigma funcional
   - 2.1 Las funciones son ciudadanos de *primer nivel*. Es decir, se pueden utilizar de las mismas maneras en las que se utiliza cualquier otro tipo de valor.
   - 2.2 El modo de trabajo de los programas realizados con **Haskell** se centra alrededor de la evaluación de expresiones en lugar de la ejecución de instrucciones.
3. Es un lenguaje **puro**
   - 3.1 Es inmutable por diseño
   - 3.2 El ser inmutable garantiza la ausencia de efectos secundarios
   - 3.3 Tiene características de idempotencia
   - 3.4 Beneficia el paralelismo
4. Es un lenguaje con evaluación tardía (lazy)

Primera prueba con Haskell

1. Abrir el navegador e ir a:
   `https://repl.it/languages/haskell`
   Plan B: `https://tio.run/#haskell`
2. Tipeamos el siguiente código

```
square(x) = x * x
main = print (square 42)
```

3. Presionamos **Run**
4. ¿Qué obtenemos?

Cómo funciona la evaluación tardía

```
square (1 + 2)          -- la expresión no se evalúa
=> (1 + 2) * (1 + 2) -- aquí ya debe comenzar a evaluarse
=> 3 * (1 + 2)
=> 3 * 3
=> 9
```

Función definida en el lenguaje

```haskell
-- http://bit.ly/haskell-and
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && x = False
```

Ejemplo:

```haskell
('H' == 'i') && ('a' == 'm')
```

Otro enfoque: ¿qué hace el siguiente código?

```java
int[] lst = {2, 3, 5, 7, 11};

int total = 0;
for (int i = 0; i < lst.length; i++)
  total = total + 3 * lst[i]; // indexitis!

System.out.println(total);
```

Mismo ejemplo, ahora con Haskell

```java
int[] lst = {2, 3, 5, 7, 11};

int total = 0;
for (int i = 0; i < lst.length; i++)
  total = total + 3 * lst[i]; // indexitis!

System.out.println(total);
```

```haskell
lst = [2, 3, 5, 7, 11]
total = sum (map (3*) lst)
main = print total
```

Funciones propias

```
sumatoria :: Int -> Int
sumatoria 0 = 0
sumatoria n = n + sumatoria (n - 1)

main = print (sumatoria 10)
```

Hailstone

```haskell
-- https://rosettacode.org/wiki/Hailstone_sequence#Haskell
hailstone :: Int -> Int
hailstone n
  | even n    = n `div` 2
  | otherwise = 3 * n + 1

main = print (hailstone 3)
```

Ejercicio: Fibonacci

?

Ejercicio: Fibonacci (Resolución)

```haskell
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)

main = print (fib 10)
```

Funciones sobre listas

```haskell
-- ¿Qué hace este código?
misterio :: [Int] -> Int
misterio [] = 0
misterio (x:xs) = 1 + misterio xs

main = print (misterio [1, 2, 3, 4, 5])
```

Funciones sobre listas

```haskell
-- ¿Y este otro?
misterioDos :: [Int] -> [Int]
misterioDos []     = []
misterioDos [x]    = [x * x]
misterioDos (x:xs) = (x * x) : (misterioDos xs)

main = print (misterioDos [1, 2, 3, 4, 5])
```

Ejercicio: Contar los pares de una lista

```
contarPares :: [Int] -> Int
```

Ejercicio: Contar los pares de una lista (resolución)

```haskell
contar :: Int -> Int
contar x = case (x `mod` 2) of
   0 -> 1
   _ -> 0

contarPares :: [Int] -> Int
contarPares [] = 0
contarPares (x:xs) = (contar x) + contarPares xs

main = print (contarPares [1, 2, 3, 4, 5, 6])
```

Quiz: *¿Cómo hacemos para sumarlos?*

Ejercicio: Sumar los pares de una lista (resolución)

```haskell
sumar :: Int -> Int
sumar x = case (x `mod` 2) of
    0 -> x
    _ -> 0

contarPares :: [Int] -> Int
contarPares [] = 0
contarPares (x:xs) = (sumar x) + contarPares xs

main = print (contarPares [1, 2, 3, 4, 5, 6])
```

Bonus: Contar “notables”

```haskell
notable :: Int -> Int
notable x = case (x `mod` 2) of
    0 -> 1
    _ -> 0

contarNotables :: (Int -> Int) -> [Int] -> Int
contarNotables f [] = 0
contarNotables f (x:xs) = (f x) + contarNotables f xs

main = print (contarNotables (notable) [1, 2, 3, 4, 5, 6])
```

Pattern Matching

```
quitaTres :: [a] -> [a]
quitaTres (_:_:_:xs) = xs
quitaTres _          = []
```

Dos operadores notables

- El operador ++ sirve para concatenar dos listas
- El operador :, en cambio, sirve para agregar elementos antes de las listas

```
x:xs     -- válido
x++xs    -- no válido

xs:ys    -- no válido
xs++ys   -- válido
```

Ejercicios:

1. Hacer funciones para manejo de Cola (`queue` / `dequeue`)
2. Hacer funciones para manejo de Pila (`push` / `pop`)

Resoluciones

```haskell
queue :: a -> [a] -> [a]
queue x xs = xs++[x]

dequeue :: [a] -> a
dequeue (x:xs) = x

push :: a -> [a] -> [a]
push x xs = x:xs

pop :: [a] -> a
pop (x:xs) = x
```

Currying

```
function x y = x + y
fun y = function 3 y
fun 2
```

Otro ejemplo

```haskell
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)

main = print (applyTwice (*3) 2)
-- ¿Cuál es el resultado?
```

FIN