Introducción a Haskell

Anabel Ovide
Ignacio Ballesteros
Luis Eduardo Bueso
Santiago Cervantes

https://github.com/edububa/haskell_course

17 de mayo de 2017

Índice

Introducción

Instalación

- Con gestores de paquetes
  - Debian/Ubuntu

```
sudo apt-get update
sudo apt-get install ghc
```

  - Arch

```
sudo pacman -S ghc
```

  - macOS

```
brew install ghc
```

- Desde la web
  https://www.haskell.org/platform/mac.html
  https://www.haskell.org/platform/windows.html
  https://www.haskell.org/downloads/linux

Introducción

Introducción Histórica

Intoducción

¿Qué es Haskell?

Primeros pasos

GHCi

- Así abrimos el intérprete de Haskell

```
\$ ghci
GHCi, version 8.0.2: http://www.haskell.org/ghc/
:? for help
Prelude>
```

- Podemos escribir expresiones aritméticas y lógicas:

```
Prelude> 2 + 2
4
Prelude> True && False
False
```

Primeros pasos

GHCi

- Podemos llamar a funciones

```
Prelude> 2 + 2
4
Prelude> True && False
False
```

- Errores

```
Prelude> 2 + "hola"

<interactive>:6:1: error:
• No instance for (Num [Char]) arising from a use
of ‘+’
• In the expression: 2 + "hola"
In an equation for ‘it’: it = 2 + "hola"
```

Primeros pasos

GHCi

- Expresiones útiles:

```
Prelude> :t 5
5 :: Num t => t
Prelude> :t 2
2 :: Num t => t
Prelude> :t "hola"
"hola" :: [Char]

Prelude> :l introduccion.hs
[1 of 1] Compiling Main
( introduccion.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Primeros pasos

Primeras funciones

Ejemplos:

```
doubleMe x = x + x

doubleUs x y = doubleMe x + doubleMe y

doubleSmallNumber x = (if x > 100 then x else x*2) + 1
```

Primeros pasos

Listas

Creando listas en un fichero .hs:

```
list = [1,2,3,4,5]
list' = [1..5]
```

Ejemplos de listas en el GHCi

```
Prelude> let list = [1,2,3,4,5]
Prelude> let list1 = [1..5]
Prelude> list == list1
list == list1
True
Prelude>
```

Primeros pasos

Listas

Las listas pueden concatenarse con el operador ++ y construirse con el operador :

```
Prelude> [1..9] ++ [10..19]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]
Prelude> "hello " ++ "world!"
"hello world!"
Prelude> 1:[2..5]
1:[2..5]
[1,2,3,4,5]
Prelude> 'h':"ello"
"hello"
Prelude> 1:[]
[1]
```

Primeros pasos

Listas

Las operaciones básicas en las listas son:

- head:

```
Prelude> head [1..10]
1
```

- tail:

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

- last:

```
Prelude> last [1..10]
10
```

Primeros pasos

Listas

- `init`:

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

- `length`:

```
Prelude> length [1..10]
10
```

- `reverse`:

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

- `take`:

```
Prelude> take 4 [1..10]
[1,2,3,4]
```

Primeros pasos

Tuplas

```
tuple = (1, False)
```

Las operaciones básicas en las tuplas son:

- fst

```
Prelude> let tuple = (1, False)
Prelude> fst tuple
1
```

- snd

```
Prelude> let tuple = (1, False)
Prelude> snd tuple
False
Prelude>
```

Tipos y *Typeclasses*

Tipos

```haskell
doubleMe :: Int -> Int
doubleMe x = x + x

doubleMe :: Int -> Int -> Int
doubleUs x y = doubleMe x + doubleMe y

doubleSmallNumber :: Int -> Int
doubleSmallNumber x = (if x > 100 then x else x*2) + 1
```

Tipos y *Typeclasses*

Typeclasses

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

doubleMe :: (Num a) => a -> a -> a
doubleUs x y = doubleMe x + doubleMe y

doubleMe :: (Num a) => a -> a
doubleSmallNumber x = (if x > 100 then x else x*2) + 1
```

Sintaxis en funciones

Pattern matching

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

```
Prelude> :l introduccion.hs
[1 of 1] Compiling Main
( introduccion.hs, interpreted )
Ok, modules loaded: Main.
*Main> sayMe 5
"Five!"
```

Sintaxis en funciones

Guards

```
compare :: (Ord a) => a -> a -> Ordering
a `compare` b
    | a > b      = GT
    | a == b     = EQ
    | otherwise = LT
```

```
Prelude> :l introduccion.hs
[1 of 1] Compiling Main
( introduccion.hs, interpreted )
Ok, modules loaded: Main.
*Main> 1 `myCompare` 2
LT
*Main> 1 `myCompare` 0.5
GT
```

Sintaxis en funciones

where

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

Sintaxis en funciones

let

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

Sintaxis en funciones

Case Expressions

```
describeList :: [a] -> String
describeList xs = "The list is " ++
    case xs of [] -> "empty."
               [x] -> "a singleton list."
               xs -> "a longer list."
```

Recursión

```haskell
factorial :: Num a => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Recursión

```haskell
duplicar :: Num a => [a] -> [a]
duplicar []     = []
duplicar [x]    = [x*2]
duplicar (x:xs) = (x*2):(duplicar xs)
```

```haskell
duplicar :: Num a => [a] -> [a]
duplicar = map ((*) 2)
```

Aclaraciones

Todo es mentira.