Haskell

- Propósito General
- Tipado Estático con inferencia de tipos
- Funcional
- Puro
- Perezoso

Instalación

GHCup

- GHC: Compilador
- Cabal: Manejador de proyectos
- HLS: Servidor de lenguaje para tu editor

Declaraciones

```
integer :: Int
integer = 3

str :: String
str = "Hello, World!"

character :: Char
character = 'C'

tuple :: (Int, String, Char)
tuple = (integer, str, character)

main :: IO ()
main = print tuple
```

Funciones

```
doubleMe :: Int -> Int
doubleMe x = 2 * x

doubleMe = \x -> 2 * x
```

Currying

Caza de Patrones

```
xor :: Bool -> Bool -> Bool
xor True False = True
xor False True = True
xor _ _ = False
```

Patrones con variables

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

Guardas y bindings

```
bmiTell :: Float -> Float -> String
bmiTell weight height
    | bmi <= 18.5 = "Rango 1"
    | bmi <= 25.0 = "Rango 2"
    | bmi <= 30.0 = "Rango 3"
    | otherwise   = "Rango 4"
    where bmi = weight / height ^ 2
```

Let bindings

```
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

Expresiones Case

```
and :: Bool -> Bool -> Bool
and y x =
  case y of
    True -> x
    False -> False
```

Polimorfismo

¿Cuál debería ser el tipo de la siguiente función?

```
id :: ? -> ?
id x = x
```

```
id :: a -> a

($) :: (a -> b) -> a -> b

(.) :: (b -> c) -> (a -> b) -> a -> c
```

Listas

```
listaVacia = []

listaNoVacia = [1, 2, 3, 4]

listaInfinita = [0, 1..]
```

Caza de Patrones

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs)
    | x > maxTail = x
    | otherwise = maxTail
    where maxTail = maximum' xs
```

- `map`
- `filter`
- `take`
- `drop`
- `length`
- `foldr`

Tipos de Datos

Sinónimos de tipos

```
type ListaNumeros = [Int]

type Matrix a = [[a]]
```

Newtype

```
newtype Pulgadas = Pulgadas Double
newtype Cm       = Cm       Double
```

Tipos Producto

```
data Rectangle = MkRectangle Float Float

rectangleArea :: Rectangle -> Float
rectangleArea (MkRectangle h w) = h * w
```

Tipos Suma

```haskell
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday

weekend :: Day -> Bool
weekend Saturday = True
weekend Sunday = True
weekend _ = False
```

En General: Tipos de datos algebráicos

```
data Shape = Rectangle Float Float | Square Float

area :: Shape -> Float
area (Rectangle h w) = h * w
area (Square s) = s * s
```

Tipos Recursivos

```haskell
data List a = Nil | Cons a (List a)

lista :: List Char
lista = Cons 'a' (Cons 'b' (Cons 'c' Nil))
```

Typeclasses

```
data List a = Nil | Cons a (List a) deriving Show
```

¿Qué hace `deriving Show`?

```
ghci> :i Show
type Show :: * -> Constraint
class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
  {-# MINIMAL showsPrec | show #-}
```

¿Y si quiero dar mi propio `show`?

```
instance Show a => Show (List a) where
    show Nil = "[]"
    show (Cons h t) = "(" ++ show h ++ " : " ++ show t ++ ")"
```

Referencias

- haskell-course
- Intro to Haskell Syntax
- Typeclasses: Polymorphism in Haskell