# Programación en Haskell

## Programación Declarativa

Maximiliano Eschoyez

2021

# El lenguaje Haskell

- ¿Por qué Haskell? $\rightarrow$ Presentación
- Historia sobre programación funcional
- Sitio oficial $\rightarrow$ www.haskell.org
- Bibliografía
  1. Documentación oficial
  2. Glasgow Haskell Compiler (GHC)
  3. Prelude Tour
  4. Learn You a Haskell for Great Good!
  5. RAZONANDO CON HASKELL. Un curso sobre programación funcional
- Breve intro para probar

# Herramientas para trabajar

GHC Glasgow Haskell Compiler

- Haskell Platform

vscode Visual Studio Code con las extensiones

- *Haskell*
- *Haskell Syntax Highlighting*

# Lenguaje Haskell

Se puede compilar

- Archivos estándar tiene extensión `.hs`
- Archivos *literate* tiene extensión `.lhs`

Se puede usar en forma interactiva

# GHCi - Entorno Interactivo

Breve introducción para usar GHCi

- `:h` o `:help` lo obvio
- Configuraciones útiles
  - `:set +t` muestra tipos de datos
  - `:set +s` muestra estadísticas de ejecución
- `:l <arch>` o `:load <arch>` carga el archivo y lo interpreta
- `:q`, `:quit` o `Ctl+D` para salir
- No se pueden usar instrucciones multilínea directamente. Se pueden escribir
  - separadas por punto y coma

```
signo :: (Integral a) => a -> a; signo x = mod x
```

  - encerradas entre llaves

```
:{
signo :: (Integral a) => a -> a
signo x = mod x 2
:}
```

# Tipos de Datos

- `Num` ⇒ Es un valor numérico
- `Real` ⇒ Es un valor numérico real
- `Fractional` ⇒ Es un valor numérico fraccional
- `Integral` ⇒ Es un valor numérico entero
  - `Int` ⇒ Limitado
  - `Integer` ⇒ Virtualmente infinito
- `Floating` ⇒ Es un valor de punto flotante
  - `Float` ⇒ Precisión simple
  - `Double` ⇒ Precisión doble
- `Bool` ⇒ Es un valor Booleano
- `Char` ⇒ Es un caracter
- `Eq` ⇒ Tiene definida la igualdad
- `Ord` ⇒ Es ordenable
- `Enum` ⇒ Es enumerable
- `Show` ⇒ Se puede mostrar como texto
- `Read` ⇒ Se puede obtener a partir de texto

# Operadores Básicos

- `+` ⇒ Suma
- `-` ⇒ Resta o cambio de signo
- `*` ⇒ Multiplicación
- `/` ⇒ División
- `div` ⇒ División entera
- `mod` ⇒ División modular
- `**` ⇒ Potencia con argumentos `Floating`
- `^` ⇒ Potencia con primer argumento `Num` y segundo `Integral`
- `%` ⇒ Simplifica la relación entre dos `Integral`

# Operadores Básicos

- `==` $\Rightarrow$ Igual
- `/=` $\Rightarrow$ Distinto
- `<`, `<=` $\Rightarrow$ Menor, menor igual
- `>`, `>=` $\Rightarrow$ Mayor, mayor igual
- `&&` $\Rightarrow$ Y lógico
- `||` $\Rightarrow$ O lógico

# Operadores y Llamado a Funciones

- Los operadores son funciones con *definición especial*
- Los operadores son *infijos*
- Se pueden cambiar a *prefijos* usando paréntesis
- Las funciones no requieren paréntesis
- Las funciones son *prefijas*
- Se pueden cambiar a *infijas* con comillas francesas ( ` )

# Definición de Funciones – *por composición*

Las funciones sin restricciones se definen de forma simple por *composición*

```
Prelude> sumaDoble x y = 2 * (x + y)
Prelude> sumaDoble 6 8
28
```

Haskell infiere tipo de dato, generalmente sin inconveniente, pero conviene indicarlos con `::` y `->`

```
Prelude> :{
Prelude| sumaDoble :: Integer -> Integer -> Integer
Prelude| sumaDoble x y = 2 * (x + y)
Prelude| :}
```

Las funciones devuelven solo un resultado, siendo el último tipo de dato el tipo del resultado y los anteriores los tipos de los argumentos

# Definición de Funciones – *por composición*

Haskell es *fuertemente tipado* y no permite aplicación de otros tipos de datos

```
Prelude> sumaDoble 6 8
28
Prelude> sumaDoble 6.5 7.5

<interactive>:21:11: error:
    * No instance for (Fractional Integer)
    arising from the literal '6.5'
    * In the first argument of 'sumaDoble', namely '6.5'
    In the expression: sumaDoble 6.5 7.5
    In an equation for 'it': it = sumaDoble 6.5 7.5
```

# Definición de Funciones – *por composición*

Haskell permite polimofismo de tipos, se usan patrones para definir la misma función para datos de la misma clase

```
Prelude> :{
Prelude| sumaDoble :: (Num a) => a -> a -> a
Prelude| sumaDoble x y = 2 * (x + y)
Prelude| :}
Prelude> sumaDoble 6 8
28
Prelude> sumaDoble 6.5 7.5
28.0
```

Para facilitar la escritura, vamos a usar un archivo y lo leeremos en el intérprete con el comando `:load` o `:l`

```
Prelude> :l factorial.lhs
[1 of 1] Compiling Main   ( factorial.lhs, interpreted )
Ok, one module loaded.
```

Ver presentación `tema_03.pdf` y `tema_04.pdf`

# Definición de Funciones – *con condicionales*

Si se necesita evaluar datos para la aplicación, la definición por composición puede hacerse *con condicionales*

```
fact' :: Int -> Int
fact' n = if n > 0 then n * fact' (n - 1)
    else if n == 0 then 1 else error "Negativo"
```

La *indentación* indica que continua el renglón anterior

# Definición de Funciones – *comparación patrones*

Una alternativa es mediante *comparación de patrones*

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n - 1)
```

Notar que esta versión no controla el ingreso de valores negativos

# Definición de Funciones – *por partes*

Una alternativa a la *comparación de patrones* es la definición *por partes*

```
fact :: Int -> Int
factorial n
    | n == 0 = 1
    | n > 0  = n * factorial (n - 1)
    | otherwise  = error "Negativo"
```

# Evaluación de Funciones

## Ansiosa – *eager evaluation*

- La más utilizada en programación
- Se resuelve lo más inmediatemente posible
- Los argumentos se evalúan y luego se pasan

## Perezosa – *lazy evaluation*

- Se demora la resolución lo más posible
- Solo se evalúa lo necesario
- Permite trabajar con representaciones infinitas
- Puede consumir más memoria
- No realiza más pasos que la evaluación ansiosa