

# Programación Funcional en Haskell

## Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales  
Universidad de Buenos Aires

1 de Febrero de 2018

# Hoy presentamos...

- 1 Tipos algebraicos
- 2 Recursión estructural en tipos algebraicos
- 3 Otros esquemas de recursión sobre listas
- 4 Generación infinita

# Tipos algebraicos y su definición en Haskell

## Tipos algebraicos

- Se definen mediante la cláusula `data`
- Tienen sus propios constructores nombrados
- Sus argumentos pueden ser **combinación de otros tipos y recursivos**

## Algunos ejemplos

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

# Tipos algebraicos y su definición en Haskell

## Tipos algebraicos

- Se definen mediante la cláusula `data`
- Tienen sus propios constructores nombrados
- Sus argumentos pueden ser **combinación de otros tipos y recursivos**

## Algunos ejemplos

```
data Maybe a = Nothing | Just a
```

```
data Either a b = Left a | Right b
```

```
data Formula = Proposicion String | No Formula  
              | Y Formula Formula  
              | O Formula Formula  
              | Imp Formula Formula
```

## Folds sobre estructuras nuevas



# Folds sobre estructuras nuevas

Dada la siguiente definición:

## Fórmulas

```
data Formula = Proposicion String | No Formula
              | Y Formula Formula
              | O Formula Formula
              | Imp Formula Formula
```

## Ejercicio 1

Definir:

- i. Esquema de recursión estructural para el tipo Formula.
- ii. proposiciones :: Formula -> [String]
- iii. quitarImplicaciones :: Formula -> Formula que convierte todas las formulas de la pinta  $(p \implies q)$  a  $(\neg p \vee q)$
- iv. evaluar :: [(String, Bool)] -> Formula -> Bool que dada una formula y los valores de verdad asignados a cada una de sus proposiciones, nos devuelve el resultado de evaluar la fórmula lógica.

# Folds sobre estructuras nuevas

Dada la siguiente definición:

RoseTree

```
data RoseTree a = Rose a [RoseTree a]
```

# Folds sobre estructuras nuevas

Dada la siguiente definición:

**RoseTree**

```
data RoseTree a = Rose a [RoseTree a]
```

## Ejercicio 2

Definir:

- i. Esquema de recursión estructural para RoseTree.
- ii. hojas, que dado un RoseTree, devuelva una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el RoseTree.
- iii. distancias, que dado un RoseTree, devuelva las distancias de su raíz a cada una de sus hojas.
- iv. altura, que devuelve la altura de un RoseTree (la cantidad de nodos de la rama más larga). Si el RoseTree es una hoja, se considera que su altura es 1.



## Otros esquemas de recursión sobre listas: Divide & Conquer

La técnica de Divide & Conquer consiste en dividir un problema en problemas más fáciles de resolver y luego combinando los resultados parciales, lograr obtener un resultado general.

Para generalizar la técnica, crearemos el tipo `DivideConquer` definido como:

```
type DivideConquer a b
```

```
= (a -> Bool)
```

```
-> (a -> b)
```

```
-> (a -> [a])
```

```
-> ([b] -> b)
```

```
-> a
```

```
-> b
```

- determina si es o no el caso trivial

- resuelve el caso trivial

- parte el problema en sub-problemas

- combina resultados

- input

- resultado

# Otros esquemas de recursión sobre listas: Divide & Conquer

La técnica de Divide & Conquer consiste en dividir un problema en problemas más fáciles de resolver y luego combinando los resultados parciales, lograr obtener un resultado general.

Para generalizar la técnica, crearemos el tipo `DivideConquer` definido como:

<code>type DivideConquer a b</code>	
<code>= (a -&gt; Bool)</code>	– determina si es o no el caso trivial
<code>-&gt; (a -&gt; b)</code>	– resuelve el caso trivial
<code>-&gt; (a -&gt; [a])</code>	– parte el problema en sub-problemas
<code>-&gt; ([b] -&gt; b)</code>	– combina resultados
<code>-&gt; a</code>	– input
<code>-&gt; b</code>	– resultado

## Ejercicio 3

- i. Definir la función `dc :: DivideConquer a b`  
`dc esTrivial resolver repartir combinar x = ...`
- ii. Definir `mergeSort :: Ord a => [a] -> [a]`  
`mergeSort = dc ...`

# Generación Infinita

## Ejercicio 4

Definir la función

`puntosDelCuadrante :: [Punto]`

Donde `Punto`, un renombre de tipos: `type Punto = (Int, Int)`

El resultado debe ser una lista (infinita) que contenga *todos* los puntos del cuadrante superior derecho del plano (sin repetir).

# Generación Infinita

## Ejercicio 4

Definir la función

```
puntosDelCuadrante :: [Punto]
```

Donde Punto, un renombre de tipos: `type Punto = (Int, Int)`

El resultado debe ser una lista (infinita) que contenga *todos* los puntos del cuadrante superior derecho del plano (sin repetir).

## De tarea

```
listasQueSuman :: Int -> [[Int]]
```

que, dado un número natural  $n$ , devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea  $n$

```
listasPositivas :: [[Int]]
```

que contenga todas las listas finitas de enteros mayores o iguales que 1.

*i? i? i? i? i? i? i? i? i? i? i? i? i?*