

# Tipos inductivos y recursión

Programación funcional

Hasta ahora vimos:

- ▶ Valores, expresiones y reducción
- ▶ Tipos, currificación y funciones de alto orden

Ejemplos:

```
data Day = Mon | Tue | Wed | Thu | Fir | Sat | Sun
```

```
data Bool = True | False
```

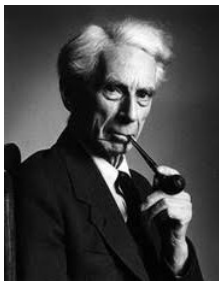
```
isWeekend :: Day -> Bool
```

```
isWeekend Sat = True
```

```
isWeekend Sun = True
```

```
isWeekend _   = False
```

# Conjuntos inductivos



Bertrand Russell

Ejemplos:

```
data Nat = Zero | Suc Nat -- 1 caso base, 1 caso inductivo
```

```
data [a] = [] | a : [a] -- 1 caso base, 1 caso inductivo
```

Propiedades:

- ▶ pueden tener infinitos elementos; pero
- ▶ todos sus elementos son finitos.

## Conjunto inductivo

Un conjunto inductivo  $S$  es aquel que puede definirse mediante dos reglas:

- ▶ Reglas base: afirman que un elemento pertenece a  $S$ ; y
- ▶ Reglas inductivas: afirman que un elemento compuesto pertenece a  $S$  si sus partes pertenecen a  $S$ .

¿Cómo podemos definir funciones sobre estos conjuntos potencialmente infinitos?

## Función recursiva

Sea  $S$  un conjunto inductivo y  $f :: S \rightarrow T$  una función. Decimos que  $f$  es *recursiva* si y sólo si:

- ▶ Se define el valor de  $(f\ x)$  para cada elemento base  $x$ ; y
- ▶ Se define el valor  $(f\ y)$  para cada elemento inductivo  $y$ , con partes inductivas  $y_1, \dots, y_n$ , en función de  $(f\ y_1), \dots, (f\ y_n)$ .

¿Qué debe cumplir una función recursiva para terminar?

1. Estar definida para los casos (base e inductivos) alcanzables.
2. Existe una función (total) monótona decreciente con respecto a los argumentos de los llamados recursivos.

## Recursión estructural

Una función recursiva está definida *estructuralmente* cuando:

- ▶ Está definida para **todos** los casos base.
- ▶ Está definida para **todos** los casos inductivos mediante una función que toma como argumentos los llamados recursivos.
- ▶ En cada llamado recursivo se “descarta” un constructor del tipo inductivo.

## Observación:

Las funciones definidas por recursión estructural siempre terminan.

# Ejemplos

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

-----

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

-----

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

-----

Cuando la llamada recursiva es la función más externa.

```
length' :: [a] -> Int
length' xs = lengthAux xs 0

lengthAux :: [a] -> Int -> Int
lengthAux [] l      = l
lengthAux (x:xs) l = lengthAux xs (l+1)
-----
```

- ▶ Puede compilarse más eficientemente (para casos en donde es imprescindible procesar la estructura entera).
- ▶ Al no ser estructural no tenemos garantía de terminación.

Cuando hay más una llamada recursiva:

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

## Desafío

¿Podemos escribir `fib` de forma lineal y a la cola?



# Tipos no-lineales

```
data Tree a = Nil | Bin a (Tree a) (Tree a)
```

```
height :: Tree -> Int
```

```
height Nil = 0
```

```
height (Bin e l r) = 1 + max (height l) (height r)
```

```
data Logic = TT | FF |
```

```
Not Logic | And Logic Logic | Or Logic Logic
```

```
value :: Logic -> Bool
```

```
value TT = True
```

```
value FF = False
```

```
value (Not p) = not (value p)
```

```
value (And p1 p2) = value p1 && value p2
```

```
value (Or p1 p2) = value p1 || value p2
```