

# Recursión

Taller de Álgebra I

Verano 2018

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.

*doble*  $x = x * 2$

# Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.

*doble*  $x = x * 2$

- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número entero?

# Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.

*doble*  $x = x * 2$

- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número entero?

$$n! = \prod_{k=1}^n k$$

# Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.

*doble*  $x = x * 2$

- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número entero?

$$n! = \prod_{k=1}^n k,$$

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si no} \end{cases}$$

# Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.

*doble*  $x = x * 2$

- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número entero?

$$n! = \prod_{k=1}^n k,$$

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si no} \end{cases}$$

¡La segunda definición de `factorial` involucra a esta misma función del lado derecho!

# Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.

*doble*  $x = x * 2$

- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número entero?

$$n! = \prod_{k=1}^n k,$$

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si no} \end{cases}$$

¡La segunda definición de `factorial` involucra a esta misma función del lado derecho!

```
factorial :: Integer -> Integer
```

# Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.

*doble*  $x = x * 2$

- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número entero?

$$n! = \prod_{k=1}^n k,$$

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si no} \end{cases}$$

¡La segunda definición de `factorial` involucra a esta misma función del lado derecho!

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



# Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.

*doble*  $x = x * 2$

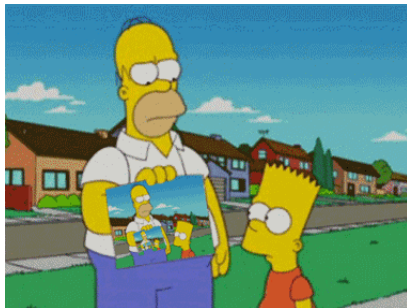
- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número entero?

$$n! = \prod_{k=1}^n k,$$

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si no} \end{cases}$$

¡La segunda definición de `factorial` involucra a esta misma función del lado derecho!

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



## ¿Cómo pensar recursivamente?

- ▶ Si queremos definir una función recursiva, por ejemplo factorial,

## ¿Cómo pensar recursivamente?

- ▶ Si queremos definir una función recursiva, por ejemplo factorial,
  - ▶ Primero, identificamos el caso base (o los casos bases). En el ejemplo de factorial, ¿Cuál era el caso base?

## ¿Cómo pensar recursivamente?

- ▶ Si queremos definir una función recursiva, por ejemplo `factorial`,
  - ▶ Primero, identificamos el caso base (o los casos bases). En el ejemplo de `factorial`, ¿Cuál era el caso base?  
En ese caso, era el valor de la función sobre 0: `factorial n | n == 0 = 1`.

## ¿Cómo pensar recursivamente?

- ▶ Si queremos definir una función recursiva, por ejemplo factorial,
  - ▶ Primero, identificamos el caso base (o los casos bases). En el ejemplo de factorial, ¿Cuál era el caso base?  
En ese caso, era el valor de la función sobre 0: `factorial n | n == 0 = 1`.
  - ▶ En el paso recursivo, **asumo** que tengo el resultado para el caso anterior, y me pregunto ¿qué falta para poder obtener el resultado que quiero?

## ¿Cómo pensar recursivamente?

- ▶ Si queremos definir una función recursiva, por ejemplo factorial,
  - ▶ Primero, identificamos el caso base (o los casos bases). En el ejemplo de factorial, ¿Cuál era el caso base?  
En ese caso, era el valor de la función sobre 0: `factorial n | n == 0 = 1`.
  - ▶ En el paso recursivo, **asumo** que tengo el resultado para el caso anterior, y me pregunto ¿qué falta para poder obtener el resultado que quiero?  
En este caso, suponemos ya calculado `factorial (n-1)` y lo combinamos multiplicándolo por `n` para lograr obtener `factorial n = n * factorial (n-1)`.

## ¿Cómo pensar recursivamente?

- ▶ Si queremos definir una función recursiva, por ejemplo factorial,
  - ▶ Primero, identificamos el caso base (o los casos bases). En el ejemplo de factorial, ¿Cuál era el caso base?  
En ese caso, era el valor de la función sobre 0: `factorial n | n == 0 = 1`.
  - ▶ En el paso recursivo, **asumo** que tengo el resultado para el caso anterior, y me pregunto ¿qué falta para poder obtener el resultado que quiero?  
En este caso, suponemos ya calculado factorial (n-1) y lo combinamos multiplicándolo por n para lograr obtener factorial `n = n * factorial (n-1)`.
- ▶ Propiedades de una definición recursiva:
  - ▶ Tiene que tener uno o más **casos base** que dependerán del tipo de llamado recursivo.  
Un caso base, es aquella expresión que no tiene paso recursivo.
  - ▶ Las **llamadas recursivas** tienen que “acercarse” a un caso base para eventualmente “terminar”.
- ▶ En cierto sentido, la recursión es el equivalente computacional de la **inducción** para las demostraciones.

## Ejercicios

- 1 Implementar la función `fib :: Integer -> Integer` que devuelve el  $i$ -ésimo número de Fibonacci. Recordar que la secuencia de Fibonacci se define como:

$$fib(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{en otro caso} \end{cases}$$

- 2 Implementar funciones recursivas para calcular el  $n$ -ésimo término de las siguientes sucesiones del Ejercicio 16 y 20 de la Práctica 2.

Para todo  $n \in \mathbb{N}$ .

1  $a_1 = 2$   
 $a_{n+1} = 2na_n + 2^{n+1}n!$

2  $a_1 = 1$   
 $a_2 = 2$   
 $a_{n+2} = na_{n+1} + 2(n+1)a_n$

3  $a_1 = -3$   
 $a_2 = 6$   
 $a_{n+2} = \begin{cases} -a_{n+1} - 3 & \text{si } n \text{ es impar} \\ a_{n+1} + 2a_n + 9 & \text{si } n \text{ es par} \end{cases}$



## Implementación de sumatorias

¿Cómo podemos implementar la función `sumatoria :: Integer -> Integer`, donde  $sumatoria(n) = \sum_{i=1}^n i$

# Sumatorias

## Implementación de sumatorias

¿Cómo podemos implementar la función *sumatoria* :: Integer -> Integer, donde  $sumatoria(n) = \sum_{i=1}^n i$

Para resolver este tipo de ejercicios, se puede pensar a las sumatorias como

$$sumatoria(n) = \sum_{i=1}^n i = n + \sum_{i=1}^{n-1} i \quad \text{para } n > 1$$

# Sumatorias

## Implementación de sumatorias

¿Cómo podemos implementar la función *sumatoria* :: Integer -> Integer, donde  $sumatoria(n) = \sum_{i=1}^n i$

Para resolver este tipo de ejercicios, se puede pensar a las sumatorias como

$$sumatoria(n) = \sum_{i=1}^n i = n + \sum_{i=1}^{n-1} i \quad \text{para } n > 1$$

Es decir:

$$sumatoria(n) = n + sumatoria(n - 1) \quad \text{para } n > 1$$

# Sumatorias

## Implementación de sumatorias

¿Cómo podemos implementar la función sumatoria :: Integer -> Integer, donde  $sumatoria(n) = \sum_{i=1}^n i$

Para resolver este tipo de ejercicios, se puede pensar a las sumatorias como

$$sumatoria(n) = \sum_{i=1}^n i = n + \sum_{i=1}^{n-1} i \quad \text{para } n > 1$$

Es decir:

$$sumatoria(n) = n + sumatoria(n - 1) \quad \text{para } n > 1$$

## Ejercicios: otras sumatorias

Implementar y dar el tipo de las siguientes funciones del Ejercicio 5 Práctica 2.

$$1 \quad f1(n) = \sum_{i=0}^n 2^i, \quad n \in \mathbb{N}_0.$$

$$3 \quad f3(n, q) = \sum_{i=1}^{2n} q^i, \quad n \in \mathbb{N}_0 \text{ y } q \in \mathbb{R}$$

$$2 \quad f2(n, q) = \sum_{i=1}^n q^i, \quad n \in \mathbb{N}_0 \text{ y } q \in \mathbb{R}$$

$$4 \quad f4(n, q) = \sum_{i=n}^{2n} \frac{q^i}{2}, \quad n \in \mathbb{N}_0 \text{ y } q \in \mathbb{R}$$

A veces el paso recursivo no es obvio o no está dado explícitamente. Hay que pensar...

### Ejercicios

- ▶ Implementar la función `esPar :: Integer -> Bool` que determine si un número natural es par. No está permitido utilizar *mod* ni *div*.
- ▶ Escribir una función para determinar si un número natural es múltiplo de 3. No está permitido utilizar *mod* ni *div*.
- ▶ Implementar la función `sumaImpares :: Integer -> Integer` que dado  $n \in \mathbb{N}$  sume los primeros  $n$  números impares. Ej: `sumaImpares 3`  $\rightsquigarrow$  `1+3+5`  $\rightsquigarrow$  `9`.
- ▶ Escribir una función `doblefact` que dado un número natural par calcula  $n!! = n(n-2)(n-4)\dots 2$ .  
Por ejemplo: `doblefact 10`  $\rightsquigarrow$  `10 * 8 * 6 * 4 * 2`  $\rightsquigarrow$  `3840`.
- ▶ Escribir una función recursiva que no termine si se la ejecuta con números negativos (y en cambio sí termine para el resto de los números).

## Asegurarse de llegar a un caso base

- Consideremos este programa recursivo para determinar si un número es par:

```
► par :: Integer -> Bool
  par n | n==0 = True
        | otherwise = par (n-2)
```

## Asegurarse de llegar a un caso base

- Consideremos este programa recursivo para determinar si un número es par:

```
► par :: Integer -> Bool
  par n | n==0 = True
        | otherwise = par (n-2)
```

¿Qué problema tiene esta función?

## Asegurarse de llegar a un caso base

- Consideremos este programa recursivo para determinar si un número es par:

```
► par :: Integer -> Bool
  par n | n==0 = True
        | otherwise = par (n-2)
```

¿Qué problema tiene esta función?

¿Cómo se arregla?



# Asegurarse de llegar a un caso base

- Consideremos este programa recursivo para determinar si un número es par:

```
► par :: Integer -> Bool
par n | n==0 = True
      | otherwise = par (n-2)
```

¿Qué problema tiene esta función?

¿Cómo se arregla?

```
► par :: Integer -> Bool
par n | n==0 = True
      | n==1 = False
      | otherwise = par (n-2)
```

```
► par :: Integer -> Bool
par n | n==0 = True
      | otherwise = not (par (n-1))
```