

Recursión

Taller de Álgebra I

Primer cuatrimestre 2019

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

Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.
- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número $n \in \mathbb{N}_0$?

Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.
- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número $n \in \mathbb{N}_0$?

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

Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.
- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número $n \in \mathbb{N}_0$?

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

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.
- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número $n \in \mathbb{N}_0$?

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

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

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

Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.
- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número $n \in \mathbb{N}_0$?

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

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

¡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”.
- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número $n \in \mathbb{N}_0$?

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

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

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

```
factorial :: Integer -> Integer
factorial n
  | n == 0 = 1
```


Recursión

- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.
- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número $n \in \mathbb{N}_0$?

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

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

¡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

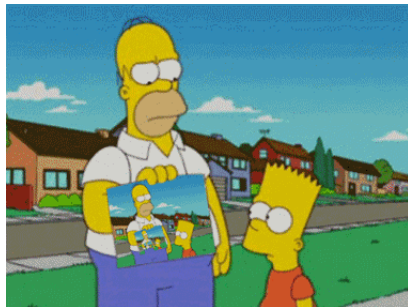
- ▶ Hasta ahora, especificamos funciones que consistían en “expresiones sencillas”.
- ▶ ¿Cómo es una función en Haskell para calcular el factorial de un número $n \in \mathbb{N}_0$?

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

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

¡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)
```



Ejercicios

1 Dada la función:

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

Reducir las expresiones:

- ▶ factorial 0
- ▶ factorial 3
- ▶ factorial (-1)

¿y si estaba definida así?:

```
factorial :: Integer -> Integer
factorial n
  | n == 0 = 1
  | otherwise = 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`,
 - ▶ en el paso recursivo, **suponiendo** que tenemos el resultado para el caso anterior, ¿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`.

¿Cómo pensar recursivamente?

- ▶ Si queremos definir una función recursiva, por ejemplo `factorial`,
 - ▶ en el paso recursivo, **suponiendo** que tenemos el resultado para el caso anterior, ¿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`.
 - ▶ además, identificamos el o los casos base. En el ejemplo de `factorial`, definimos como casos base la función sobre 0: `factorial n | n == 0 = 1`

¿Cómo pensar recursivamente?

- ▶ Si queremos definir una función recursiva, por ejemplo factorial,
 - ▶ en el paso recursivo, **suponiendo** que tenemos el resultado para el caso anterior, ¿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.
 - ▶ además, identificamos el o los casos base. En el ejemplo de factorial, definimos como casos base la función sobre 0: `factorial n | n == 0 = 1`
- ▶ Propiedades de una definición recursiva:
 - ▶ las **llamadas recursivas** tienen que “acercarse” a un caso base.
 - ▶ 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.

¿Cómo pensar recursivamente?

- ▶ Si queremos definir una función recursiva, por ejemplo factorial,
 - ▶ en el paso recursivo, **suponiendo** que tenemos el resultado para el caso anterior, ¿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.
 - ▶ además, identificamos el o los casos base. En el ejemplo de factorial, definimos como casos base la función sobre 0: `factorial n | n == 0 = 1`
- ▶ Propiedades de una definición recursiva:
 - ▶ las **llamadas recursivas** tienen que “acercarse” a un caso base.
 - ▶ 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.
- ▶ En cierto sentido, la recursión es el equivalente computacional de la **inducción** para las demostraciones.

Asegurarse de llegar a un caso base

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

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

Asegurarse de llegar a un caso base

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

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

¿Qué problema tiene esta función?

Asegurarse de llegar a un caso base

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

```
esPar :: Integer -> Bool
esPar n | n==0 = True
        | otherwise = esPar (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 entero positivo es par:

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

¿Qué problema tiene esta función?

¿Cómo se arregla?

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

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

Ejercicios

- 2 Implementar la función $fib : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{Z}$ que devuelve el i -ésimo número de Fibonacci. Recordar que la secuencia de Fibonacci se define como:

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

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

a $a_1 = 2$, $a_{n+1} = 2n a_n + 2^{n+1}n!$, para todo $n \in \mathbb{N}$.

Pueden verificar con el término general $a_n = 2^n n!$

b $a_1 = 1$, $a_2 = 2$ y $a_{n+2} = n a_{n+1} + 2(n+1)a_n$, para todo $n \in \mathbb{N}$.

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

Implementación de sumatorias

¿Cómo podemos implementar la función *sumatoria* : $\mathbb{N} \rightarrow \mathbb{N}$, donde $sumatoria(n) = \sum_{i=1}^n i$

Implementación de sumatorias

¿Cómo podemos implementar la función *sumatoria* : $\mathbb{N} \rightarrow \mathbb{N}$, donde

$$\text{sumatoria}(n) = \sum_{i=1}^n i$$

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

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

Sumatorias

Implementación de sumatorias

¿Cómo podemos implementar la función *sumatoria* : $\mathbb{N} \rightarrow \mathbb{N}$, donde

$$\text{sumatoria}(n) = \sum_{i=1}^n i$$

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

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

Es decir:

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

Sumatorias

Implementación de sumatorias

¿Cómo podemos implementar la función *sumatoria* : $\mathbb{N} \rightarrow \mathbb{N}$, donde

$$\text{sumatoria}(n) = \sum_{i=1}^n i$$

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

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

Es decir:

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

Ejercicios: otras sumatorias

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

$$\text{16 } f1(n) = \sum_{i=0}^n 2^i, \quad n \in \mathbb{N}_0.$$

$$\text{18 } f3(n, q) = \sum_{i=1}^{2n} q^i, \quad n \in \mathbb{N}_0 \text{ y } q \in \mathbb{R}$$

$$\text{17 } f2(n, q) = \sum_{i=1}^n q^i, \quad n \in \mathbb{N} \text{ y } q \in \mathbb{R}$$

$$\text{19 } f4(n, q) = \sum_{i=n}^{2n} q^i, \quad n \in \mathbb{N}_0 \text{ y } q \in \mathbb{R}$$

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

Ejercicios

- 20 Escribir una función para determinar si un número natural es múltiplo de 3. No está permitido utilizar `mod` ni `div`.
- 21 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`.
- 22 Escribir una función `medioFact` que dado $n \in \mathbb{N}$ calcula $n!! = n(n-2)(n-4)\dots$. Por ejemplo:
`medioFact 10` \rightsquigarrow `10 * 8 * 6 * 4 * 2` \rightsquigarrow `3840`.
`medioFact 9` \rightsquigarrow `9 * 7 * 5 * 3 * 1` \rightsquigarrow `945`.
- 23 Escribir una función recursiva que no termine si se la ejecuta con enteros negativos (y en cambio sí termine para el resto de los enteros).