Mecanismo de Reducción

Taller de Álgebra I

Primer cuatrimestre 2018

Ejercicios de números enteros

Ejercicios

Dar el tipo y luego implementar las siguientes funciones:

1. `unidades`: dado un entero, devuelve el dígito de las unidades del número (el dígito menos significativo).
2. `sumaUnidades3`: dados 3 enteros, devuelve la suma de los dígitos de las unidades de los 3 números.
3. `todosImpares`: dados 3 números enteros determina si son todos impares.
4. `alMenosUnImpar`: dados 3 números enteros determina si al menos uno de ellos es impar.
5. `alMenosDosImpares`: dados 3 números enteros determina si al menos dos de ellos son impares.
6. `alMenosDosPares`: dados 3 números enteros determina si al menos dos de ellos son pares.
7. `alMenosUnMultiploDe`: dados 3 números enteros determina si alguno de los primeros dos es múltiplo del tercero

Acerca del último ejercicio

Podemos definir `alMenosUnMultiploDe` utilizando las siguientes funciones:

```
esMultiploDe :: Integer -> Integer -> Bool
esMultiploDe x y = mod x y == 0

alMenosUnMultiploDe :: Integer -> Integer -> Integer -> Bool
alMenosUnMultiploDe x y n = (esMultiploDe x n) || (esMultiploDe y n)
```

Acerca del último ejercicio

Podemos definir `alMenosUnMultiploDe` utilizando las siguientes funciones:

```
esMultiploDe :: Integer -> Integer -> Bool
esMultiploDe x y = mod x y == 0

alMenosUnMultiploDe :: Integer -> Integer -> Integer -> Bool
alMenosUnMultiploDe x y n = (esMultiploDe x n) || (esMultiploDe y n)
```

¿Qué pasa si trato de evaluar `alMenosUnMultiploDe 4 5 0` ?

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).
- ¿Cómo podemos clasificar las funciones?

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).
- ¿Cómo podemos clasificar las funciones?
  - Funciones totales: nunca se indefinen.

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).
- ¿Cómo podemos clasificar las funciones?
  - Funciones totales: nunca se indefinen.

```
suc ::  Integer -> Integer
suc x = x + 1
```

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).
- ¿Cómo podemos clasificar las funciones?
  - Funciones totales: nunca se indefinen.

```
suc ::  Integer -> Integer
suc x = x + 1
```

  - Funciones parciales: hay argumentos para los cuales se indefinen.

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).
- ¿Cómo podemos clasificar las funciones?
  - Funciones totales: nunca se indefinen.

```
suc ::  Integer -> Integer
suc x = x + 1
```

  - Funciones parciales: hay argumentos para los cuales se indefinen.

```
inv ::  Float -> Float
inv x | x /= 0 = 1/x
```

Ejercicios de relaciones

Ejercicios

8. Dados dos enteros a, b implementar funciones:
   ($r1$, $r2$ y $r3$) `::` `Integer` `->` `Integer` `->` `Bool` que determinen si $a \sim b$ donde:
   1. $a \sim b$ si tienen la misma paridad
   2. $a \sim b$ si $2a + 3b$ es divisible por 5
   3. $a \sim b$ si los dígitos de las unidades de a, b y $a \cdot b$ son todos distintos
9. Se define en $\mathbb{R}$ la relación de equivalencia asociada a la partición

   $$\mathbb{R} = (-\infty, 3) \cup [3, +\infty)$$

   Determinar el tipo e implementar una función que dados dos números $x, y \in \mathbb{R}$ determine si $x \sim y$.
10. Repetir el ejercicio anterior para la partición

    $$\mathbb{R} = (-\infty, 3) \cup [3, 7) \cup [7, +\infty).$$

11. Dados (a, b) y (p, q), determinar el tipo e implementar funciones que determinen si $(a, b) \sim (p, q)$ cuando:
    1. $(a, b), (p, q) \in \mathbb{R}_{\neq 0} \times \mathbb{R}_{\neq 0}$, $(a, b) \sim (p, q)$ si $\exists k \in \mathbb{R}$ tal que $(a, b) = k(p, q)$
    2. $(a, b), (p, q) \in \mathbb{Z}_{\neq 0} \times \mathbb{Z}_{\neq 0}$, $(a, b) \sim (p, q)$ si existe $k \in \mathbb{R}$ tal que $(a, b) = k(p, q)$
    3. (Opcional) $(a, b), (p, q) \in \mathbb{Z} \times \mathbb{Z} - \{(0, 0)\}$ $(a, b) \sim (p, q)$ si existe $k \in \mathbb{R}$ tal que $(a, b) = k(p, q)$

Un ejercicio más

- $(a, b), (p, q) \in \mathbb{R} \times \mathbb{R}$, $(a, b) \sim (p, q)$ si $p \neq 0, q \neq 0$ y $\exists k \in \mathbb{R}$ tal que $(a, b) = k(p, q)$

Una solución puede ser:

```
ej11 :: (Float,Float) -> (Float,Float) -> Bool
ej11 (a,b) (p,q) | p==0 || q==0 = False
                 | otherwise    = a/p == b/q
```

Un ejercicio más

- $(a, b), (p, q) \in \mathbb{R} \times \mathbb{R}$, $(a, b) \sim (p, q)$ si $p \neq 0, q \neq 0$ y $\exists k \in \mathbb{R}$ tal que $(a, b) = k(p, q)$

Una solución puede ser:

```
ej11 :: (Float,Float) -> (Float,Float) -> Bool
ej11 (a,b) (p,q) | p==0 || q==0 = False
                 | otherwise    = a/p == b/q
```

Consideremos la siguiente alternativa:

```
ej11a :: (Float,Float) -> (Float,Float) -> Bool
ej11a (a,b) (p,q) = (p/=0 && q/=0) && a/p == b/q
```

¿Debería funcionar la evaluación de `ej10a (4,5) (0,0)`?

Reducción

Problema $\rightarrow$ Algoritmo $\rightarrow$ Programa

Reducción

Problema $\rightarrow$ Algoritmo $\rightarrow$ Programa

¿Cómo ejecuta Haskell?

¿Qué sucede en Haskell si escribo una expresión? ¿Cómo se transforma esa expresión en un resultado?

Reducción

Problema $\rightarrow$ Algoritmo $\rightarrow$ Programa

¿Cómo ejecuta Haskell?

¿Qué sucede en Haskell si escribo una expresión? ¿Cómo se transforma esa expresión en un resultado?

- Dado el siguiente programa:

```
resta :: Integer -> Integer -> Integer
resta x y = x - y

suma :: Integer -> Integer -> Integer
suma x y = x + y

negar :: Integer -> Integer
negar x = -x
```

- ¿Qué sucede al evaluar la expresión `suma (resta 2 (negar 42)) 4`

Reducción

```
suma (resta 2 (negar 42)) 4
```

- El mecanismo de evaluación en un Lenguaje Funcional es la reducción:

Reducción

```
suma (resta 2 (negar 42)) 4
```

- El mecanismo de evaluación en un Lenguaje Funcional es la reducción:
  1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.

Reducción

```
suma (resta 2 (negar 42)) 4
```

- El mecanismo de evaluación en un Lenguaje Funcional es la reducción:
  1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
  2. La subexpresión a reemplazar es alguna instancia del lado izquierdo de alguna ecuación orientada del programa, y se la llama radical o redex (*reducible expression*).
     - Buscamos un redex: `suma` (`resta 2 (negar 42)`) `4`

redex

Reducción

```
suma (resta 2 (negar 42)) 4
```

- El mecanismo de evaluación en un Lenguaje Funcional es la reducción:
  1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
  2. La subexpresión a reemplazar es alguna instancia del lado izquierdo de alguna ecuación orientada del programa, y se la llama radical o redex (*reducible expression*).
     - Buscamos un redex: `suma` (`resta 2 (negar 42)`) `4`

       redex
  3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.
     - `resta x y = x - y`
     - `x` $\leftarrow$ `2`
     - `y` $\leftarrow$ `(negar 42)`

Reducción

```
suma (resta 2 (negar 42)) 4
```

- El mecanismo de evaluación en un Lenguaje Funcional es la reducción:
  1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
  2. La subexpresión a reemplazar es alguna instancia del lado izquierdo de alguna ecuación orientada del programa, y se la llama radical o redex (*reducible expression*).
     - Buscamos un redex: `suma` (`resta 2 (negar 42)`) `4`

       redex
  3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.
     - `resta x y = x - y`
     - `x` $\leftarrow$ `2`
     - `y` $\leftarrow$ `(negar 42)`
  4. Reemplazamos el redex con lo anterior y el resto de la expresión no cambia.
     - `suma (resta 2 (negar 42)) 4` $\rightsquigarrow$ `suma (2 - (negar 42)) 4`

Reducción

```
suma (resta 2 (negar 42)) 4
```

- El mecanismo de evaluación en un Lenguaje Funcional es la reducción:
  1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
  2. La subexpresión a reemplazar es alguna instancia del lado izquierdo de alguna ecuación orientada del programa, y se la llama radical o redex (*reducible expression*).
     - Buscamos un redex: `suma` (`resta 2 (negar 42)`) `4`

       redex
  3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.
     - `resta x y = x - y`
     - `x` $\leftarrow$ `2`
     - `y` $\leftarrow$ `(negar 42)`
  4. Reemplazamos el redex con lo anterior y el resto de la expresión no cambia.
     - `suma (resta 2 (negar 42)) 4` $\rightsquigarrow$ `suma (2 - (negar 42)) 4`
  5. Si la expresión resultante aún puede reducirse, volvemos al paso 1.

Órdenes de evaluación de expresiones

Una expresión puede ser evaluada en orden eager ("ansioso") o lazy ("perezoso"):

- Si se evalúa en orden eager, en una expresión que contiene una función y sus argumentos, primero se evalúan los argumentos y luego la función.
- El orden de evaluación lazy reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

La reducción de expresiones en Haskell es lazy.

Órdenes de evaluación de expresiones

Una expresión puede ser evaluada en orden eager ("ansioso") o lazy ("perezoso"):

- Si se evalúa en orden eager, en una expresión que contiene una función y sus argumentos, primero se evalúan los argumentos y luego la función.
- El orden de evaluación lazy reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

La reducción de expresiones en Haskell es lazy.

Ejemplo:

```
suma (3+4) (suc (2*3))
```

Órdenes de evaluación de expresiones

Una expresión puede ser evaluada en orden eager ("ansioso") o lazy ("perezoso"):

- Si se evalúa en orden eager, en una expresión que contiene una función y sus argumentos, primero se evalúan los argumentos y luego la función.
- El orden de evaluación lazy reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

La reducción de expresiones en Haskell es lazy.

Ejemplo:

```
 suma (3+4) (suc (2*3))
⇝  (3+4) + (suc (2*3))
```

Órdenes de evaluación de expresiones

Una expresión puede ser evaluada en orden eager ("ansioso") o lazy ("perezoso"):

- Si se evalúa en orden eager, en una expresión que contiene una función y sus argumentos, primero se evalúan los argumentos y luego la función.
- El orden de evaluación lazy reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

La reducción de expresiones en Haskell es lazy.

Ejemplo:

```
 suma (3+4) (suc (2*3))
⇝  (3+4) + (suc (2*3))
⇝  7 + (suc (2*3))
```

Órdenes de evaluación de expresiones

Una expresión puede ser evaluada en orden eager ("ansioso") o lazy ("perezoso"):

- Si se evalúa en orden eager, en una expresión que contiene una función y sus argumentos, primero se evalúan los argumentos y luego la función.
- El orden de evaluación lazy reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

La reducción de expresiones en Haskell es lazy.

Ejemplo:

```
 suma (3+4) (suc (2*3))
⇝  (3+4) + (suc (2*3))
⇝  7 + (suc (2*3))
⇝  7 + ((2*3) + 1)
```

Órdenes de evaluación de expresiones

Una expresión puede ser evaluada en orden eager ("ansioso") o lazy ("perezoso"):

- Si se evalúa en orden eager, en una expresión que contiene una función y sus argumentos, primero se evalúan los argumentos y luego la función.
- El orden de evaluación lazy reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

La reducción de expresiones en Haskell es lazy.

Ejemplo:

```
 suma (3+4) (suc (2*3))
⇝  (3+4) + (suc (2*3))
⇝  7 + (suc (2*3))
⇝  7 + ((2*3) + 1)
⇝  7 + (6 + 1)
```

Órdenes de evaluación de expresiones

Una expresión puede ser evaluada en orden eager ("ansioso") o lazy ("perezoso"):

- Si se evalúa en orden eager, en una expresión que contiene una función y sus argumentos, primero se evalúan los argumentos y luego la función.
- El orden de evaluación lazy reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

La reducción de expresiones en Haskell es lazy.

Ejemplo:

```
 suma (3+4) (suc (2*3))
⇝ (3+4) + (suc (2*3))
⇝ 7 + (suc (2*3))
⇝ 7 + ((2*3) + 1)
⇝ 7 + (6 + 1)
⇝ 7 + 7
```

Órdenes de evaluación de expresiones

Una expresión puede ser evaluada en orden eager ("ansioso") o lazy ("perezoso"):

- Si se evalúa en orden eager, en una expresión que contiene una función y sus argumentos, primero se evalúan los argumentos y luego la función.
- El orden de evaluación lazy reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

La reducción de expresiones en Haskell es lazy.

Ejemplo:

```
 suma (3+4) (suc (2*3))
⇝ (3+4) + (suc (2*3))
⇝ 7 + (suc (2*3))
⇝ 7 + ((2*3) + 1)
⇝ 7 + (6 + 1)
⇝ 7 + 7
⇝ 14
```

Volviendo al ejemplo anterior

```
ej10a :: (Float,Float) -> (Float,Float) -> Bool
ej10a (a,b) (p,q) = (p/=0 && q/=0) && a/p == b/q
```

Evaluamos `ej10a (4,5) (0,0)`:

```
ej10a (4,5) (0,0)
```

Volviendo al ejemplo anterior

```
ej10a :: (Float,Float) -> (Float,Float) -> Bool
ej10a (a,b) (p,q) = (p/=0 && q/=0) && a/p == b/q
```

Evaluamos `ej10a (4,5) (0,0)`:

```
 ej10a (4,5) (0,0)
⇝  (0/=0 && 0/=0) && 5/0 == 4/0
```

Volviendo al ejemplo anterior

```
ej10a :: (Float,Float) -> (Float,Float) -> Bool
ej10a (a,b) (p,q) = (p/=0 && q/=0) && a/p == b/q
```

Evaluamos `ej10a (4,5) (0,0)`:

```
 ej10a (4,5) (0,0)
⇝  (0/=0 && 0/=0) && 5/0 == 4/0
⇝  (False && 0/=0) && 5/0 == 4/0
```

Volviendo al ejemplo anterior

```
ej10a :: (Float,Float) -> (Float,Float) -> Bool
ej10a (a,b) (p,q) = (p/=0 && q/=0) && a/p == b/q
```

Evaluamos `ej10a (4,5) (0,0)`:

```
 ej10a (4,5) (0,0)
⇝  (0/=0 && 0/=0) && 5/0 == 4/0
⇝  (False && 0/=0) && 5/0 == 4/0
⇝ False && 5/0 == 4/0
```

Volviendo al ejemplo anterior

```
ej10a :: (Float,Float) -> (Float,Float) -> Bool
ej10a (a,b) (p,q) = (p/=0 && q/=0) && a/p == b/q
```

Evaluamos `ej10a (4,5) (0,0)`:

```
 ej10a (4,5) (0,0)
⇝ (0/=0 && 0/=0) && 5/0 == 4/0
⇝ (False && 0/=0) && 5/0 == 4/0
⇝ False && 5/0 == 4/0
⇝ False
```

Recursión

- 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 entero?

Recursión

- Hasta ahora, especificamos funciones que consistían en "expresiones sencillas".
- ¿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".
- ¿Cómo es una función en Haskell para calcular el factorial de un número entero?

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

Recursión

- Hasta ahora, especificamos funciones que consistían en "expresiones sencillas".
- ¿Cómo es una función en Haskell para calcular el factorial de un número entero?

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

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

Recursión

- Hasta ahora, especificamos funciones que consistían en "expresiones sencillas".
- ¿Cómo es una función en Haskell para calcular el factorial de un número entero?

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

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

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

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

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

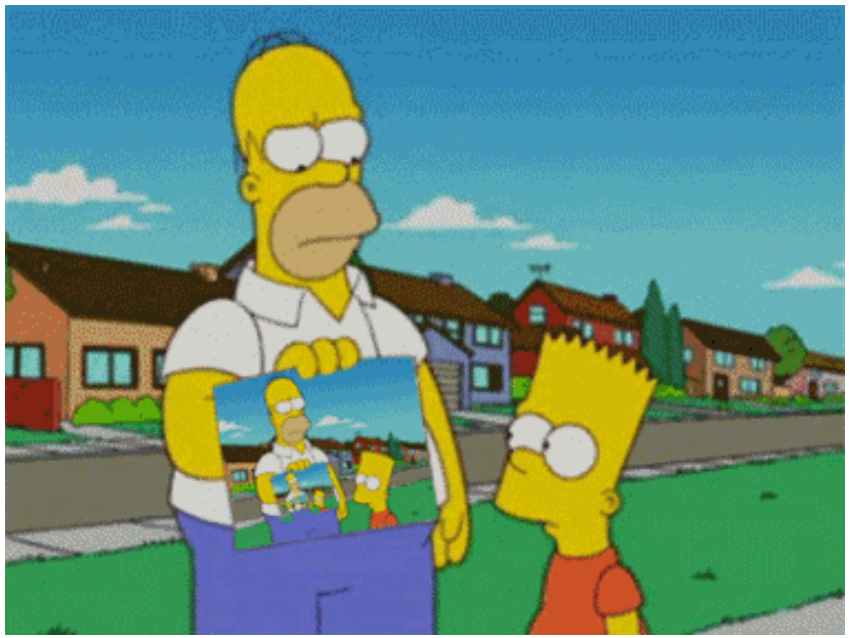
Recursión

- Hasta ahora, especificamos funciones que consistían en "expresiones sencillas".
- ¿Cómo es una función en Haskell para calcular el factorial de un número entero?

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

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

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



Ejercicios

1. Implementar la función `sc :: Integer -> Integer` definida por

$$sc(n) = \begin{cases} 0 & \text{si } n = 0 \\ sc(n-1) + n^2 & \text{si no} \end{cases}$$

2. Implementar la función `fib :: Integer -> Integer` que devuelve el i-ésimo número de Fibonacci. Recordar que la sucesión 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.
   1. $a_1 = 2$, $a_{n+1} = 2na_n + 2^{n+1}n!$, para todo $n \in \mathbb{N}$.
   2. $a_1 = 1$, $a_2 = 2$ y $a_{n+2} = na_{n+1} + 2(n+1)a_n$, para todo $n \in \mathbb{N}$.
   3. $a_1 = -3$, $a_2 = 6$ y $a_{n+2} = \begin{cases} -a_{n+1} - 3 & \text{si } n \text{ es impar} \\ a_{n+1} + 2a_n + 9 & \text{si } n \text{ es par} \end{cases}$

Definiciones recursivas

Ojo

¡Si una función recursiva no se define correctamente, tendrá valor $\perp$!

Definiciones recursivas

Ojo

¡Si una función recursiva no se define correctamente, tendrá valor $\bot$!

¿Cómo pensar recursivamente?

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

Definiciones recursivas

Ojo

¡Si una función recursiva no se define correctamente, tendrá valor $\bot$!

¿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`.

Definiciones recursivas

Ojo

¡Si una función recursiva no se define correctamente, tendrá valor $\perp$!

¿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`

Definiciones recursivas

Ojo

¡Si una función recursiva no se define correctamente, tendrá valor $\perp$!

¿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.

Definiciones recursivas

Ojo

¡Si una función recursiva no se define correctamente, tendrá valor $\perp$!

¿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.