# Reducción y Recursión

Taller de Álgebra I

Segundo Cuatrimestre de 2016

# Reducción

- Dado el siguiente programa:

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

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

-- Funcion que devuelve la cantidad de digitos de x
digitos :: Integer -> Integer
digitos x = ??
```

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

# Reducción

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

- El mecanismo de evaluación en un Lenguaje Funcional es la reducción:
  1. Vamos a reemplazar una 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: $\texttt{suma}\ (\underbrace{\texttt{resta 2 (digitos 42)}}_{\text{redex}})\ \texttt{4}$
  3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.
     - `resta x y = x - y`
     - $\texttt{x} \leftarrow \texttt{2}$
     - $\texttt{y} \leftarrow \texttt{(digitos 42)}$
  4. Reemplazamos el redex con lo anterior y el resto de la expresión no cambia.
     - $\texttt{suma (resta 2 (digitos 42)) 4} \rightsquigarrow \texttt{suma (2 - (digitos 42)) 4}$

# Órdenes de evaluación en Haskell

Orden normal o lazy (“perezoso”):

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).

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
```

# Indefinición

- Las expresiones que 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
    ```

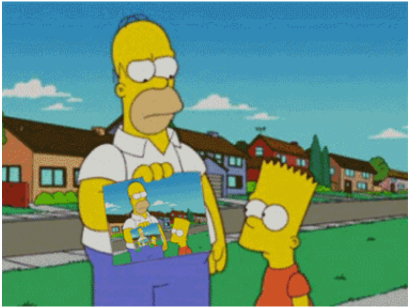
# Recursión

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

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

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

# Definiciones recursivas



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

# Definiciones recursivas

- Propiedades de una definición recursiva:

  1. Tiene que tener uno o más casos base. Un caso base, es aquella expresión que no tiene paso recursivo.
  2. Las llamadas recursivas del lado derecho tienen que *acercarse* al caso base, con relación a los parámetros del lado izquierdo de la ecuación.

- En cierto sentido, la recursión es el equivalente computacional de la inducción para las demostraciones.

# Otras funciones recursivas

## Programar las siguientes funciones

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

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

- Implementar la función `par :: Integer -> Bool` que determine si un número es par. 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 para determinar si un número es múltiplo de 3. No está permitido utilizar mod ni div.

# Asegurarse de llegar a un caso base

- Consideremos este programa recursivo para determinar si un número es par:
- ```
  par ::  Integer -> Bool
  par 0 = True
  par n = par (n-2)
  ```

  ¿Qué problema tiene esta función?

  ¿Cómo se arregla?
- ```
  par 0 = True
  par 1 = False
  par n = par (n-2)
  ```
- ```
  par 0 = True
  par n = not (par (n-1))
  ```
  ```

## ¿Cómo pensar recursivamente?

- Si tenemos la función `n!`
- Primero hay que pensar sobre qué se está haciendo recursión. Por ejemplo, en este caso, la recursión se hace sobre los enteros no negativos.
- Identificamos el o los casos base.
  En el ejemplo de `n!`, definimos como casos base el 0: `0!= 1`
- En el paso recursivo, suponiendo que tenemos el resultado de la llamada recursiva, ¿qué falta para poder obtener el resultado que quiero?
  En este caso, suponemos calculado `(n-1)!` y lo combinamos multiplicándolo por `n`.

# Ejercicios

1. Escribir una función `doblefact` para calcular $n!! = n\,(n-2)(n-4)\ldots 2$.
   Por ejemplo: `doblefact 10` $\rightsquigarrow 10*8*6*4*2 \rightsquigarrow 3840$.
   La función se debe indefinir para los números impares.
2. Escribir una función que dados $n$, $m \in \mathbb{N}$ compute el combinatorio $\binom{n}{m}$. Hacerlo usando la igualdad $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$.
3. 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).
4. Escribir una función que dado $n \in \mathbb{N}$ sume los números impares positivos cuyo cuadrado sea menor que $n$. Por ejemplo: `sumaImparesCuyoCuadSeaMenorQue 30` $\rightsquigarrow 1+3+5 \rightsquigarrow 9$.