# Mecanismo de Reducción

Taller de Álgebra I

Verano 2018

# Repaso: ¿Qué vimos hasta ahora?

## ¿Programar para qué?

- Implementar algoritmos que solucionen problemas.
- Pensamiento algorítmico $\rightarrow$ Ver los problemas desde otro punto de vista.

---

- Definición de funciones básicas
- Funciones partidas
- Funciones con notación infija / sufija: `(+)` vs `+`, `` `mod` `` vs `mod`

---

- Vimos que tanto las funciones como las expresiones tienen TIPO.
- Algunos tipos de datos:
  - `Integer`, `Float` y `Bool`
  - Pares: `(A, B)` (para A y B tipos)
  - Tipos genéricos (con o sin restricciones):
    - `a -> a`
    - `a -> b`
    - `Num a => a -> b`
    - `(Num a, Num b) => a -> b`
- El tipo no varía con la ejecución: no es necesario ejecutar para determinar el tipo

# Reducción

## ¿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:
  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 (*re*ducible *ex*pression).
     - Buscamos un redex: $\texttt{suma}\ (\underbrace{\texttt{resta 2 (negar 42)}}_{\text{redex}})\ \texttt{4}$
  3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.
     - `resta x y = x - y`
     - `x` ← `2`
     - `y` ← `(negar 42)`
  4. Reemplazamos el redex con lo anterior y el resto de la expresión no cambia.
     - `suma (resta 2 (negar 42)) 4` ⇝ `suma (2 - (negar 42)) 4`
  5. Si la expresión resultante aún puede reducirse, volvemos al paso 1.

# 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 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
    ```

## Reducir

- `(inv 1 == 0) && (inv 0 == 1)`
- `(inv 1 == 1) && (inv 0 == 1)`
- `(inv 0 == 1) && (inv 1 == 1)`

# Ejercicios de números enteros

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.

# Ejercicios de relaciones

7. 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 $ab$ son todos distintos
8. 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 $x \sim y$.
9. Repetir el ejercicio anterior para la partición
   $$\mathbb{R} = (-\infty, 3) \cup [3, 7) \cup [7, +\infty).$$
10. Dados $(a, b)$ y $(p, q)$ en $\mathbb{Z} \times \mathbb{Z} - \{(0, 0)\}$, determinar el tipo e implementar funciones que determinen si $(a, b) \sim (p, q)$ para las siguientes relaciones:
    1. $(a, b) \sim (p, q)$ si existe $k \in \mathbb{Z}$ tal que $(a, b) = k(p, q)$
    2. $(a, b) \sim (p, q)$ si existe $k \in \mathbb{R}$ tal que $(a, b) = k(p, q)$

# Ejercicios de números complejos

Representamos a los números complejos $a + bi$ como tuplas $(a, b)$, con $a$ y $b$ de tipo `Float`.

11. Implementar la función que suma dos números complejos:

```
sumaC :: (Float, Float) -> (Float, Float) -> (Float, Float)
sumaC z w = ??
```

12. Determinar el tipo e implementar las siguientes funciones:

    1. `productoC`: producto de dos números complejos
    2. `productoPorRealC`: producto de un número real por un número complejo
    3. `conjugadoC`: conjugado de un número complejo
    4. `inversoC`: inverso de un número complejo (no nulo)
       Recordar: $\frac{1}{a+bi} = \frac{a-bi}{a^2+b^2}$ si $a^2 + b^2 \neq 0$.
       Recomendación: utilizar las funciónes `conjugadoC` y `productoPorRealC` ya definidas.

13. Implementar la función

```
raices :: Float -> Float -> Float -> ((Float, Float), (Float, Float))
```

que dados los números reales $a$, $b$ y $c$ devuelve las dos raíces del polinomio $ax^2 + bx + c$ como par de números complejos.

Sugerencia: distinguir los casos del discriminante $b^2 - 4ac \geq 0$ y $b^2 - 4ac < 0$.

14. Utilizar la función anterior para calcular las raíces de $2x^2 + 4x - 6 = 0$ y $x^2 - 4x + 5 = 0$.