

# Introducción a la Programación Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2023

Introducción a la Programación Funcional

# Algoritmos y programas

- ▶ Hasta ahora estudiamos lógica y aprendimos a **especificar** problemas
- ▶ El objetivo es ahora escribir un **algoritmo** que cumpla esa especificación
  - ▶ Secuencia de pasos que pueden llevarse a cabo mecánicamente
- ▶ Puede haber varios algoritmos que cumplan una misma especificación
- ▶ Una vez que se tiene el algoritmo, se escribe el **programa** que implementa el algoritmo
  - ▶ Expresión formal de un algoritmo
  - ▶ Lenguajes de programación
    - ▶ sintaxis definida
    - ▶ semántica definida
    - ▶ qué hace una computadora cuando recibe ese programa
    - ▶ qué especificaciones cumple
    - ▶ ejemplos: Haskell, C, C++, C#, Python, Java, Smalltalk, Prolog, etc.
- ▶ A partir de un algoritmo van a existir múltiples programas que implementan dicho algoritmo.

# Paradigmas de Programación

- ▶ Existen distintos paradigmas de programación
  - ▶ Formas de pensar un algoritmo que cumpla una especificación
  - ▶ Cada uno tiene asociado un conjunto de lenguajes
  - ▶ Nos llevan a encarar la programación según ese paradigma
- ▶ Haskell pertenece al paradigma de programación funcional
  - ▶ programa = colección de funciones
    - ▶ Transforman datos de entrada en un resultado
  - ▶ Los lenguajes funcionales nos dan herramientas para explicarle a la computadora cómo computar esas funciones

# Programación funcional

- ▶ Un **programa** en un language funcional es un **conjunto de ecuaciones orientadas** que definen una o más funciones.

Por ejemplo:

```
doble x = x + x
```

- ▶ La **ejecución** de un programa en este caso corresponde a la **evaluación de una expresión**, habitualmente solicitada desde la consola del entorno de programación.

```
Prelude> doble 10  
20
```

- ▶ La expresión se evalúa usando las ecuaciones definidas en el programa, hasta llegar a un resultado.
- ▶ Las ecuaciones orientadas junto con el mecanismo de reducción describen **algoritmos**.

# Ecuaciones

Para determinar el valor de la aplicación de una función se reemplaza cada expresión por otra, según las ecuaciones.

- ▶ Este proceso puede no terminar, aún con ecuaciones bien definidas.
- ▶ Por ejemplo, consideremos la expresión:

`doble (1 + 1)`

Si reemplazamos `1 + 1` por `doble 1` obtenemos `doble (doble 1)`

Y ahora podemos reemplazar `doble 1` por `1 + 1`

Volvimos a empezar...

`doble (1 + 1)  $\rightsquigarrow$  doble (doble 1)  $\rightsquigarrow$  doble (1 + 1)  $\rightsquigarrow$  ...`

# Ecuaciones orientadas

- ▶ Lado **izquierdo**: expresión a definir
- ▶ Lado **derecho**: definición
- ▶ Cálculo del valor de una expresión : reemplazamos las subexpresiones que sean lado izquierdo de una ecuación por su lado derecho

Ejemplo: `doble x = x + x`

`doble (1 + 1)  $\rightsquigarrow$  (1 + 1) + (1 + 1)  $\rightsquigarrow$  2 + (1 + 1)  $\rightsquigarrow$  2 + 2  $\rightsquigarrow$  4`

También podría ser:

`doble (1 + 1)  $\rightsquigarrow$  doble 2  $\rightsquigarrow$  2 + 2  $\rightsquigarrow$  4`

Más adelante veremos cómo funciona Haskell en particular.

# Transparencia referencial

Es la propiedad de un lenguaje que garantiza que el valor de una expresión depende exclusivamente de sus subexpresiones.

Por lo tanto,

- ▶ Cada expresión del lenguaje representa siempre el mismo valor en cualquier lugar de un programa
- ▶ Es una propiedad muy importante en el paradigma de la programación funcional.
  - ▶ En otros paradigmas el significado de una expresión depende del contexto
- ▶ Es muy útil para verificar correctitud (demostrar que se cumple la especificación)
  - ▶ Podemos usar propiedades ya probadas para sub expresiones
  - ▶ El valor no depende de la historia
  - ▶ Valen en cualquier contexto

# Formación de expresiones

## ► Expresiones **atómicas**

- También se llaman **formas normales**
- Son las más simples, no se puede **reducir** más.
- Son la forma más intuitiva de representar un valor
- Ejemplos:
  - 2
  - False
  - (3, True)
- Es común llamarlas “valores” aunque no son un valor, *denotan* un valor, como las demás expresiones

## ► Expresiones **compuestas**

- Se construyen combinando expresiones atómicas con operaciones
- Ejemplos:
  - 1+1
  - 1==2
  - (4-1, True || False)



# Formación de expresiones

- ▶ Algunas cadenas de símbolos no forman expresiones
  - ▶ por problemas sintácticos:
    - ▶ `+*1-`
    - ▶ `(True`
    - ▶ `('a',)`
  - ▶ o por error de tipos:
    - ▶ `2 + False`
    - ▶ `2 || 'a'`
    - ▶ `4 * 'b'`
- ▶ Para saber si una expresión está bien formada, aplicamos
  - ▶ Reglas sintácticas
  - ▶ Reglas de asignación o inferencia de tipos (algoritmo de Hindley-Milner)
- ▶ En Haskell toda expresión denota un valor, y ese valor pertenece a un tipo de datos y no se puede usar como si fuera de otro tipo distinto.
  - ▶ Haskell es un lenguaje **fuertemente tipado**

# ¿Cómo ejecuta Haskell?

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

► Dado el siguiente programa:

```
resta x y = x - y
```

```
suma x y = x + y
```

```
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** (*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 ← 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  $\rightsquigarrow$  suma (2 - (negar 42)) 4

5. Si la expresión resultante aún puede reducirse, volvemos al paso 1, sino llegamos a una expresión atómica (forma normal) y ese es el resultado del cómputo.

suma (2 - (negar 42)) 4  $\rightsquigarrow$  suma (2 - (- 42)) 4  
suma (2 - (- 42)) 4  $\rightsquigarrow$  suma (44)  
4 suma (44) 4  $\rightsquigarrow$  44 + 4  $\rightsquigarrow$  48

# Órdenes de evaluación en Haskell

Haskell tiene un orden de **evaluación normal** o **lazy** (perezoso): se reduce el redex más externo y más a la izquierda para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero se 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
```

Otros lenguajes de programación (C, C++, Pascal, Java) tienen un orden de **evaluación eager** (ansioso): primero se evalúan los argumentos y después la función.

# 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 x = x + 1`
  - ▶ Funciones **parciales**: hay argumentos para los cuales se indefinen.  
`division x y = div x y`

¿Qué pasa al reducir las siguientes expresiones en Haskell?

- ▶ `(division 1 1 == 0) && (division 1 0 == 1)`
- ▶ `(division 1 1 == 1) && (division 1 0 == 1)`
- ▶ `(division 1 0 == 1) && (division 1 1 == 1)`

¿Y si hiciéramos una evaluación eager o ansiosa?

# Definiciones de funciones por casos

Podemos usar **guardas** para definir funciones por casos:

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

```
f n | n == 0 = 1  
    | n /= 0 = 0
```

Palabra clave “si no”.

```
f n | n == 0 = 1  
    | otherwise = 0
```

# La función signo

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

```
signo n | n > 0 = 1  
        | n == 0 = 0  
        | n < 0 = -1
```

```
signo n | n > 0 = 1  
        | n == 0 = 0  
        | otherwise = -1
```

# La función máximo

```
maximo x y | x >= y = x  
           | otherwise = y
```



# ¿Qué hacen las siguientes funciones?

```
f1 n | n >= 3 = 5
```

```
f2 n | n >= 3 = 5  
    | n <= 1 = 8
```

```
f3 n | n >= 3 = 5  
    | n == 2 = undefined  
    | otherwise = 8
```

# ¿Qué hacen las siguientes funciones?

```
f4 n | n >= 3 = 5  
      | n <= 9 = 7
```

```
f5 n | n <= 9 = 7  
      | n >= 3 = 5
```

**Prestar atención al orden de las guardas.** ¡Cuando las condiciones se solapan, el orden de las guardas cambia el comportamiento de la función!

## Otra posibilidad usando *pattern matching*

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

```
f n | n == 0 = 1  
    | n /= 0 = 0
```

También se puede hacer:

```
f 0 = 1  
f n = 0
```

## Otra posibilidad usando *pattern matching*

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

```
signo n | n > 0 = 1  
        | n == 0 = 0  
        | n < 0 = -1
```

También se puede hacer:

```
signo 0 = 0  
signo n | n > 0 = 1  
        | otherwise = -1
```

# Un ejemplo con especificación

Dados tres números  $a$ ,  $b$  y  $c$ , calcular la cantidad de soluciones reales de la ecuación cuadrática:  $aX^2 + bX + c = 0$ .

```
problema cantidadDeSoluciones(a :  $\mathbb{Z}$ , b :  $\mathbb{Z}$ , c :  $\mathbb{Z}$ ) :  $\mathbb{Z}$  {  
  requiere: {a  $\neq$  0}  
  asegura: {(res = 2  $\leftrightarrow$  discriminante(a, b, c) > 0)  $\vee$  (res = 1  $\leftrightarrow$   
discriminante(a, b, c) = 0)  $\vee$  (res = 0  $\leftrightarrow$  discriminante(a, b, c) < 0)}  
}  
problema discriminante(a :  $\mathbb{Z}$ , b :  $\mathbb{Z}$ , c :  $\mathbb{Z}$ ) :  $\mathbb{Z}$  {  
  requiere: {a  $\neq$  0}  
  asegura: {res =  $b^2 - 4 * a * c$ }  
}
```

```
cantidadDeSoluciones a b c | b^2 - 4*a*c > 0 = 2  
                           | b^2 - 4*a*c == 0 = 1  
                           | otherwise = 0
```

Otra posibilidad:

```
cantidadDeSoluciones a b c | discriminante > 0 = 2  
                           | discriminante == 0 = 1  
                           | otherwise = 0  
                           where discriminante = b^2 - 4*a*c
```

# Tipos de datos

Un **conjunto de valores** a los que se les puede aplicar un **conjunto de funciones**.

## Ejemplos:

1.  $\text{Int} = (\mathbb{Z}, \{+, -, *, \text{div}, \text{mod}\})$  es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.
  2.  $\text{Float} = (\mathbb{Q}, \{+, -, *, /\})$  es el tipo de datos que representa a los racionales, con la aritmética de **punto flotante**.
  3.  $\text{Char} = (\{'a', 'A', '1', '?'\}, \{\text{ord}, \text{chr}, \text{isUpper}, \text{toUpper}\})$  es el tipo de datos que representan los caracteres.
  4.  $\text{Bool} = (\{\text{True}, \text{False}\}, \{\&\&, ||, \text{not}\})$  representa a los valores lógicos.
- Podemos declarar explícitamente el tipo de datos del *dominio* y *codominio* de las funciones. A esto lo llamamos dar la **signatura** de la función.
  - No es estrictamente necesario hacerlo (Haskell puede inferir el tipo), pero suele ser una buena práctica (y **¡nosotros lo vamos a pedir!**).

# Aplicación de funciones

En programación funcional (como en matemática) las funciones son elementos (valores).

Notación  $f :: T1 \rightarrow T2 \rightarrow T3 \rightarrow \dots \rightarrow Tn$

- ▶ Una función es un valor
- ▶ la operación básica que podemos realizar con ese valor es la **aplicación**
  - ▶ Aplicar la función a un elemento para obtener un resultado
- ▶ Sintácticamente, la aplicación se escribe como una yuxtaposición (la función seguida de su parámetro).
- ▶ Por ejemplo: sea  $f :: T1 \rightarrow T2$ , y  $e$  de tipo  $T1$  entonces  $f\ e$  es una expresión de tipo  $T2$ .  
Sea  $\text{doble} :: \text{Int} \rightarrow \text{Int}$ , entonces  $\text{doble } 2$  representa un número entero.

# Ejemplos de funciones con la signatura

```
maximo :: Int -> Int -> Int
maximo x y | x >= y = x
           | otherwise = y
```

```
maximoRac :: Float -> Float -> Float
maximoRac x y | x >= y = x
              | otherwise = y
```

```
esMayorA9 :: Int -> Bool
esMayorA9 n | n > 9 = True
            | otherwise = False
```

```
esPar :: Int -> Bool
esPar n | mod n 2 == 0 = True
        | otherwise = False
```

```
esPar2 :: Int -> Bool
esPar2 n = mod n 2 == 0
```



## Otro ejemplo más raro:

```
funcionRara :: Float -> Float -> Bool -> Bool
funcionRara x y z = (x >= y) || z
```

Otras posibilidades, usando *pattern matching*:

```
funcionRara :: Float -> Float -> Bool -> Bool
funcionRara x y True = True
funcionRara x y False = x >= y
```

```
funcionRara :: Float -> Float -> Bool -> Bool
funcionRara _ _ True = True
funcionRara x y False = x >= y
```

# Polimorfismo

- ▶ Se llama polimorfismo a una función que puede aplicarse a distintos tipos de datos (sin redefinirla).
- ▶ se usa cuando el comportamiento de la función no depende paramétricamente del tipo de sus argumentos
- ▶ lo vimos en el lenguaje de especificación con las funciones genéricas.
- ▶ En Haskell los polimorfismos se escriben usando **variables de tipo** y conviven con el tipado fuerte.
- ▶ Ejemplo de una función polimórfica: la función identidad.

# Variables de tipos

¿Qué tipo tienen las siguientes funciones?

```
identidad x = x
```

```
primero x y = x
```

```
segundo x y = y
```

```
constante5 x y z = 5
```

## Variables de tipo

- ▶ Son parámetros que se escriben en la signatura usando variables minúsculas
- ▶ En lugar de valores, denotan tipos
- ▶ Cuando se invoca la función se usa como argumento el tipo del valor

# Variables de tipo (cont.)

## Funciones con variables de tipo

```
identidad :: t -> t
identidad x = x

primero :: tx -> ty -> tx
primero x y = x

segundo :: tx -> ty -> ty
segundo x y = y

constante5 :: tx -> ty -> tz -> Int
constante5 x y z = 5

mismoTipo :: t -> t -> Bool
mismoTipo x y = True
```

Si dos argumentos deben tener el mismo tipo, se debe usar la misma variable de tipo

- Luego, primero `True 5 :: Bool`, pero mismoTipo 1 `True 0` no tipa

# Clases de tipos

¿Qué tipo tienen las siguientes funciones?

```
triple x = 3*x
```

```
maximo x y | x >= y = x  
          | otherwise = y
```

```
distintos x y = x /= y
```

## Clases de tipos

- ▶ Conjunto de tipos a los que se le pueden aplicar ciertas funciones
- ▶ Un tipo puede pertenecer a distintas clases

Los `Float` son números (`Num`), con orden (`Ord`), de punto flotante (`Floating`), etc.

## En este curso

- ▶ No vamos a evaluar el uso de clases de tipos, pero ...
- ▶ ...saber la mecánica permite comprender los mensajes del compilador de Haskell (GHCi)

# Clases de tipos (cont)

## Clase de tipos

- **Conjunto de tipos de datos** a los que se les puede aplicar un **conjunto de funciones**

## Algunas clases:

1. `Integral := ({ Int, Integer, ... }, { mod, div, ... })`
2. `Fractional := ({ Float, Double, ... }, { (/), ... })`
3. `Floating := ({ Float, Double, ... }, {  
sqrt, sin, cos, tan, ... })`
4. `Num := ({ Int, Integer, Float, Double, ... }, {  
(+), (*), abs, ... })`
5. `Ord := ({ Bool, Int, Integer, Float, Double, ... }, {  
(<=), compare })`
6. `Eq := ({ Bool, Int, Integer, Float, Double, ... }, { (==), (/=)  
})`

# Clases de tipos (cont)

Las clases de tipos se describen como restricciones sobre variables de tipos

```
triple :: (Num t) => t -> t
triple x = 3*x
```

```
maximo :: (Ord t) => t -> t -> t
maximo x y | x >= y = x
           | otherwise = y
```

```
distintos :: (Eq t) => t -> t -> Bool
distintos x y = x /= y
```

— Cantidad de raíces de la ecuación:  $ax^2 + bx + c$

```
cantidadDeSoluciones :: (Num t, Ord t) => t -> t -> t -> Int
cantidadDeSoluciones a b c | discriminante > 0 = 2
                           | discriminante == 0 = 1
                           | otherwise = 0
                           where discriminante = b^2 - 4*a*c
```

```
pepe :: (Floating t, Eq t, Num u, Eq u) => t -> t -> u -> Bool
pepe x y z = sqrt (x + y) == x && 3*z == 0
```

`(Floating t, Eq t, Num u, Eq u) => ...` significa que:

- ▶ la variable `t` tiene que ser de un tipo que pertenezca a `Floating` y `Eq`
- ▶ la variable `u` tiene que ser de un tipo que pertenezca a `Num` y `Eq`

# Ejercitación conjunta

Averiguar el tipo asignado por Haskell a las siguientes funciones

```
f1 x y z = x**y + z <= x+y**z
```

```
f2 x y = (sqrt x) / (sqrt y)
```

```
f3 x y = div (sqrt x) (sqrt y)
```

```
f4 x y z | x == y = z  
          | x ** y == y = x  
          | otherwise = y
```

```
f5 x y z | x == y = z  
          | x ** y == y = z  
          | otherwise = z
```

¿Qué error ocurre cuándo ejecutamos `f4 5 5 True`? ¿Tiene sentido?

¿Y si ejecutamos `f5 5 5 True`? ¿Qué cambió?



# Nueva familia de tipos: Tuplas

## Tuplas

- Dados tipos  $A_1, \dots, A_k$ , el **tipo  $k$ -upla**  $(A_1, \dots, A_k)$  es el conjunto de las  $k$ -uplas  $(v_1, \dots, v_k)$  donde  $v_i$  es de tipo  $A_i$

```
(1, 2)           :: (Int, Int)
(1.1, 3.2, 5.0)  :: (Float, Float, Float)
(True, (1, 2))   :: (Bool, (Int, Int))
(True, 1, 2)     :: (Bool, Int, Int)
```

- En Haskell hay infinitos tipos de tuplas

## Funciones de acceso a los valores de un par en `Prelude`

- `fst`  $:: (a, b) \rightarrow a$       Ejemplo: `fst (1 + 4, 2)  $\rightsquigarrow$  5`
- `snd`  $:: (a, b) \rightarrow b$       Ejemplo: `snd (1, (2, 3))  $\rightsquigarrow$  (2, 3)`

Ejemplo: suma de vectores en  $\mathbb{R}^2$

```
suma :: (Float, Float) -> (Float, Float) -> (Float, Float)
suma v w = ((fst v) + (fst w), (snd v) + (snd w))
```

Podemos usar *pattern matching* para acceder a los valores de una tupla

```
suma (vx, vy) (wx, wy) = (vx + wx, vy + wy)
```

# Pattern matching sobre tuplas

Podemos usar *pattern matching* sobre constructores de tuplas y números

```
esOrigen :: (Float, Float) -> Bool
esOrigen (0, 0) = True
esOrigen (_, _) = False
```

```
angulo0 :: (Float, Float) -> Bool
angulo0 (_, 0) = True
angulo0 (_, _) = False
```

```
{-
No podemos usar dos veces la misma variable
angulo45 :: (Float, Float) -> Bool
angulo45 (x,x) = True
angulo45 (_,_) = False
-}
```

```
angulo45 :: (Float, Float) -> Bool
angulo45 (x,y) = x == y
```

```
patternMatching :: (Float, (Bool, Int), (Bool, (Int, Float))) -> (Float, (Int,
    Float))
patternMatching (f1, (True, -), (_, (0, f2))) = (f1, (1, f2))
patternMatching (-, -, (-, (-, f))) = (f, (0, f))
```

# Parámetros vs. tuplas

¿Conviene tener dos parámetros escalares o un parámetro dupla?

```
suma :: (Float, Float) -> (Float, Float) -> (Float, Float)
suma (vx, vy) (wx, wy) = (vx + wx, vy + wy)
```

```
— normaVectorial2 x y es la norma de (x,y)
normaVectorial2 :: Float -> Float -> Float
normaVectorial2 x y = sqrt (x^2 + y^2)
```

```
— normaVectorial1 (x,y) es la norma de (x,y)
normaVectorial1 :: (Float, Float) -> Float
normaVectorial1 (x,y) = sqrt (x^2 + y^2)
```

```
normalSuma :: (Float, Float) -> (Float, Float) -> Float
normalSuma v1 v2 = normaVectorial1 (suma v1 v2)
```

```
norma2Suma :: (Float, Float) -> (Float, Float) -> Float
norma2Suma v1 v2 = normaVectorial2 (fst s) (snd s)
  where s = suma v1 v2
```

# Curricación

- ▶ Diferencia entre promedio1 y promedio2
  - ▶ `promedio1 :: (Float, Float) -> Float`  
`promedio1 (x,y) = (x+y)/2`
  - ▶ `promedio2 :: Float -> Float -> Float`  
`promedio2 x y = (x+y)/2`
- ▶ solo cambia el tipo de datos de la función
  - ▶ promedio1 recibe un solo parámetro (una dupla)
  - ▶ promedio2 recibe dos Float separados por un espacio
  - ▶ para declararla, separamos los tipos de los parámetros con una flecha
  - ▶ tiene motivos teóricos y prácticos (que no veremos ahora)
- ▶ la notación se llama **curricación** en honor al matemático Haskell B. Curry
- ▶ para nosotros, alcanza con ver que evita el uso de varios signos de puntuación (comas y paréntesis)
  - ▶ `promedio1 (promedio1 (2, 3), promedio1 (1, 2))`
  - ▶ `promedio2 (promedio2 2 3) (promedio2 1 2)`

# Funciones binarias: notación prefija vs. infija

## Funciones binarias

- ▶ Notación prefija: función antes de los argumentos (e.g., suma x y)
- ▶ Notación infija: función entre argumentos (e.g. x + y, 5 \* 3, etc)
- ▶ La notación infija se permite para funciones cuyos nombres son operadores
- ▶ El nombre real de una función definido por un operador • es (•)
- ▶ Se puede usar el nombre real con notación prefija, e.g. (+) 2 3
- ▶ Haskell permite definir nuevas funciones con símbolos, e.g., (\*+) (no hacerlo!)
- ▶ Una función binaria f puede ser usada de forma infija escribiendo 'f'

## Ejemplos:

```
(>=) :: Ord a => a -> a -> Bool
(>=) 5 3 —evalua a True
(==) :: Eq a => a -> a -> Bool
(==) 3 4 —evalua a False
(^) :: (Num a, Int b) => a -> b -> a
(^) 2 5 —evalua 32.0
mod :: (Integral a) => a -> a -> a
5 'mod' 3 —evalua 2
div :: (Integral a) => a -> a -> a
5 'div' 3 —evalua 1
```