

Taller de Álgebra I

Clase 2 - Primeras funciones II

Primer cuatrimestre 2022

Conceptos fundamentales de las clases previas

- ▶ Definición de algoritmo
- ▶ Programa funcional: ecuaciones dirigidas + reglas evaluación
 - <función a definir> = <expresión más simple>
- ▶ Funciones en haskell
 - ▶ Definición y evaluación
 - ▶ Funciones partidas y guardas
 - ▶ Pattern matching
- ▶ Tipos de datos y signatura
 - ▶ Int: números enteros
 - ▶ Float: números “reales” (punto flotante)
 - ▶ Bool: valores de verdad (True, False)

```
f :: Float -> Float    signo :: Int -> Int    beta :: Bool -> Int
f x = x+2              signo x | x > 0      = 1    beta False = 0
                        | x == 0      = 0    beta _      = 1
                        | otherwise = -1
```

Variables de tipos

¿Qué tipo tienen las siguientes funciones y expresiones?

```
identidad x = x  
primero x y = x  
segundo x y = y  
constante5 x y z = 5  
primero True 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

- 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 de 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  $x^2 + bx + c$ 
cantidadDeSoluciones :: (Num t, Ord t) => t -> t -> Int
cantidadDeSoluciones b c | d > 0 = 2
                        | d == 0 = 1
                        | otherwise = 0
    where d = b^2 - 4*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
```

$(\text{Floating } t, \text{Eq } t, \text{Num } u, \text{Eq } u) \Rightarrow \dots$ 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) -> a` Ejemplo: `fst (1 + 4, 2) ~> 5`
- ▶ `snd :: (a, b) -> b` Ejemplo: `snd (1, (2, 3)) ~> (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)

norma1Suma :: (Float, Float) -> (Float, Float) -> Float
norma1Suma 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
```

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 \bullet es (\bullet)
- ▶ 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 común 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
```

Ejercicios

Implementar las siguientes funciones, especificando su signatura. Usar pares para representar vectores. **Opcional:** chequear qué tipo asigna Haskell cuando se elimina la signatura y argumentar las razones de cada tipo.

- 1 `estanRelacionados`: dados dos números reales, decide si están relacionados considerando la relación de equivalencia en \mathbb{R} cuyas clases de equivalencia son:

$$(-\infty, 3], (3, 7] \text{ y } (7, \infty).$$

- 2 `prodInt`: calcula el producto interno entre dos vectores de \mathbb{R}^2 .
- 3 `todoMenor`: dados dos vectores de \mathbb{R}^2 , decide si es cierto que cada coordenada del primer vector es menor a la coordenada correspondiente del segundo vector.
- 4 `distanciaPuntos`: calcula la distancia entre dos puntos de \mathbb{R}^2 .
- 5 `sumaTerna`: dada una terna de enteros, calcula la suma de sus tres elementos.
- 6 `posicPrimerPar`: dada una terna de enteros, devuelve la posición del primer número par si es que hay alguno, y devuelve 4 si son todos impares.
- 7 `crearPar :: a -> b -> (a, b)`: crea un par a partir de sus dos componentes dadas por separado (debe funcionar para elementos de cualquier tipo).
- 8 `invertir :: (a, b) -> (b, a)`: invierte los elementos del par pasado como parámetro (debe funcionar para elementos de cualquier tipo).

Desde acá hasta el final de esta clase es opcional

Lenguajes de programación como herramientas de comunicación

- ▶ Entre seres humanos
- ▶ Con la computadora

Código como mensaje a ser decodificado por un humano

- ▶ Bien ordenado, declarativo, explicado, conciso, documentado, etc.

`x == 0` vs. `sqrt (4 * fromInteger ((abs x) * (sgn x))) == 0`

Compartir código

- ▶ ¿Con quién? → otros usuarios, docentes, estudiantes, **conmigo**
- ▶ ¿Para qué? → reutilizar, testear, mantener, mejorar, **explicar**, etc.
- ▶ ¿Cómo? copy&paste (stackoverflow), herramientas externas (git) o internas (**módulos**)

En este curso:

- ▶ Conocer la mecánica de módulos para entender mensajes de error y evitar problemas

Creación de módulos en haskell

- ▶ Módulo: conjunto de identificadores con su implementación.
Identificador: nombre unívoco de alguna entidad (función, tipo, módulo, etc)
- ▶ Los módulos se declaran con la palabra reservada `module`
- ▶ Cada módulo se identifica con un nombre en mayúscula
- ▶ El módulo se escribe en un único archivo del mismo nombre con extension “.hs”
- ▶ Los identificadores internos se implementan después de la palabra reservada `where`
- ▶ Las bibliotecas y programas de escala industrial suelen contener muchos módulos

```
module ModuloBase
where

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

    divide :: Int -> Int -> Bool
    divide x y = mod y x == 0

    -- Mas y mas identificadores
```


Importación de módulos

Importación de módulos en haskell

- ▶ Con la palabra reservada `import`
- ▶ Cada módulo se importa en una línea aparte
- ▶ Los identificadores importados se usan como los no importados
- ▶ Se puede elegir qué identificadores importar entre paréntesis
- ▶ Solo se importan los módulos listados y `Prelude`; si un módulo importado M importa a su vez a un módulo N , entonces los identificadores de N no son importados.

```
module ModuloMedio
where
    import ModuloBase
    import ModuloBase2(absoluto)

    maximo3 :: Int -> Int -> Int -> Int
    maximo3 x y z = maximo (maximo x y) z

    maximoAbsoluto :: Int -> Int -> Int
    maximoAbsoluto x y = maximo (abs x) (absoluto y)
```

Ambigüedades y alcance de los identificadores

Alcance de los identificadores (scope)

- ▶ Lugares donde un identificador es válido, i.e., nombre una entidad existente
 - ▶ Comportamiento de `where` para funciones y módulos
 - ▶ Ambigüedad: un módulo no puede usar el mismo identificador para dos entidades
- En caso de ambigüedad, hay que prefijar el nombre del módulo.

```
module ModuloMedio2
where
  import ModuloBase
  import ModuloBase2
  import ModuloBase3

  maximo3 :: Int -> Int -> Int -> Int
  maximo3 x y z = ModuloBase.maximo (ModuloBase.maximo x y) z

  maximoCuadrado :: Float -> Float -> Float
  maximoCuadrado x y = ModuloBase3.maximo (x^2) (y^2)
```

Función privada

Funciones privadas

- ▶ Es común definir identificadores auxiliares en los módulos
- ▶ Los identificadores auxiliares son internos y pueden cambiar:
Desapareciendo, cambiando de tipo, propósito, nombre, etc.
- ▶ Los identificadores auxiliares no deberían usarse en otros módulos
- ▶ En Haskell, podemos seleccionar qué identificadores se exportan
- ▶ Los que no se exportan son identificadores **privados**

```
module ModuloBase2(absoluto)
where

    absoluto :: Int -> Int
    absoluto x | x >= 0 = 0
               | otherwise = negativo x

    negativo :: Int -> Int
    negativo x = -x
```

Comandos GHCi para uso de módulos

Comandos del intérprete para trabajo con módulos

- ▶ En GHCi siempre hay un módulo cargado, inicialmente `Prelude`
- ▶ Para cargar otro módulo se usa el comando `:load <Módulo>`
- ▶ Para recargar el módulo actual después de modificar se usa `:reload`
- ▶ Todos los módulos importados por el módulo principal se cargan automáticamente
- ▶ Podemos ver los identificadores de un módulo cargado con `:browse`
- ▶ Los identificadores pueden tener documentación propia que se accede con `:doc`

Pero hay que activar la documentación con `:set -haddock`

- ▶ En caso de no recordar comandos, puedo usar `:help` para averiguarlos
- ▶ Ejemplo, puedo ver que `:type` indica el tipo de una función

Documentando nuestros módulos

Escribiendo documentación

- ▶ Comentario: texto para comunicación humano-humano, ignorado por la computadora
- ▶ Una línea: todo lo que aparece después de `--`
- ▶ Multilínea: todo lo que aparece entre `{- comentario -}`
- ▶ Documentación: comentario antes de una función que empieza con `-- |` o `{- |`

Recordar activar la documentación con `:set -haddock`

```
module Documentacion
where
    {- este modulo muestra como documentar y comentar
    este comentario multilinea se ignora -}
    --este comentario tambien se ignora

    -- |Documentacion de id
    -- id es la identidad
    id x = x

    {- |Documentacion multilinea
    constante5 es la funcion constante 5-}
    constante5 = 5
```

Comandos útiles del intérprete

- ▶ `:show path` → muestra los directorios de los que se cargan módulos
- ▶ `:cd` → cambia de directorio
- ▶ `:show modules` → muestra los módulos cargados actualmente
- ▶ `:set editor <editor>` → permite definir un editor
- ▶ `:edit <file>` → permite editar/crear un archivo
- ▶ `:edit` → edita el módulo actual
- ▶ `:info <id>` → muestra información de un identificador (muy útil!)
- ▶ `:type +d <expr>` → tipo con valores por default para las variables de tipo (muy util!)
- ▶ `:set +s` → muestra el tiempo transcurrido para evaluar una función
- ▶ `:set +t` → muestra el tipo luego de cada evaluación
- ▶ `:unset` → limpia cualquier selección de set (editor, tiempo, tipo, etc)