Taller de Álgebra I

Clase 1 - Primeras funciones

Programación funcional

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

  Por ejemplo:

```
doble x = 2 * 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 (definición de los pasos para resolver un problema).

Para hacer ahora...

1. Crear un directorio específico y trabajar toda la clase ahí adentro (y todas las clases del resto del curso también).
2. Abrir el intérprete de Haskell.
3. Ejecutar alguna operación simple, por ejemplo `8 * 7`.
4. Crear un archivo de texto llamado `Clase01.hs`.
5. Escribir dentro del archivo `f x y = x * x + y * y` y guardarlo.
6. Cargar el archivo: `:l Clase01.hs`.
7. Dentro de GHCI, ejecutar lo siguiente: `f 2 3`
8. Agregar al código la función `g x y z = x + y + z * z` y volver a guardar.
9. En `ghci`, recargar el programa: `:r`
10. Ejecutar `g 2 3 4`
11. Si quieren, pueden cerrar el intérprete ejecutando: `:q`

¡Manos a la obra! Programar las siguientes funciones

- $doble(x) = 2x$
- $suma(x, y) = x + y$
- $||(x_1, x_2)|| = \sqrt{x_1^2 + x_2^2}$
- $f(x) = 8$

- `doble x = ??`
- `suma x y = ??`
- `normaVectorial x1 x2 = ??`
- `funcionConstante8 x = ??`

Ejecutar las siguientes expresiones en el intérprete

```
*Main> doble 10
*Main> doble (-1)
*Main> doble -1
*Main> suma 3 4
*Main> suma(3, 4)
*Main> suma (-1) 4
*Main> normaVectorial 3 5
*Main> funcionConstante8 0
*Main> suma (doble 10) 5
```

Machete

- Números
  - `1` (números)
  - `1.3, 1e-10, 6.022140857e23` (números con decimales)
  - `(-1)` (números negativos)
- Funciones básicas
  - `+, *, /, -`
  - `div, mod` (cociente y resto en la división entera)
  - `sqrt, **, ^`
- Uso de funciones
  - Para aplicar una función, utilizamos el nombre de la función seguido de parametros con espacios entre medio:
    - `f x1 x2 x3 x4 x5 x6`
    - Equivalente a $f(x_1, x_2, x_3, x_4, x_5, x_6)$ en matemática
    - Ejemplo: `sqrt 4`
    - Ejemplo: `div 2 3`
  - Para indicar que un parámetro es resultado de otra operación, usamos paréntesis:
    - `f (g x1) (h x2 x3) x3 x4 x5 x6`
    - Equivalente a $f(g(x_1), h(x_2, x_3), x_3, x_4, x_5, x_6)$ en matemática
    - Ejemplo: `sqrt ((sqrt 10) - 3)`
    - Ejemplo: `div (mod 3 5) (mod 4 3)`

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

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

```
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*

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

```
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
```

Otro ejemplo

Implementar la función `cantidadDeSoluciones`, que dados dos números b y c, calcula la cantidad de soluciones reales la ecuación cuadrática

$$X^2 + bX + c = 0.$$

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

Otra posibilidad:

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

Tipos de datos

Tipo de dato

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

Ejemplos

1. $\texttt{Int} = (\mathbb{Z}, \{+, -, *, \texttt{div}, \texttt{mod}\})$ es el tipo de datos que representa a los enteros con las operaciones aritméticas habituales.
2. $\texttt{Float} = (\mathbb{Q}, \{+, -, *, /\})$ es el tipo de datos que representa a los racionales, con la aritmética de punto flotante.
3. $\texttt{Bool} = (\{\texttt{True}, \texttt{False}\}, \{\&\&, ||, \texttt{not}\})$ representa a los valores lógicos.

Podemos especificar explícitamente el tipo de datos del *dominio* y *codominio* de las funciones que definimos. A esto lo llamamos dar la *signatura* de la función.

No es estrictamente necesario hacerlo, pero suele ser una buena práctica.

Ejemplos

```
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
```

```
esImpar :: Int -> Bool
esImpar n = not (esPar n)
```

Otro ejemplo mas 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
```

Ejercicios

Implementar las siguientes funciones, especificando su signatura.

1. `absoluto`: calcula el valor absoluto de un número entero.
2. `maximoabsoluto`: devuelve el máximo entre el valor absoluto de dos números enteros.
3. `maximo3`: devuelve el máximo entre tres números enteros.
4. `algunoEs0`: dados dos números racionales, decide si alguno de los dos es igual a 0 (hacerlo dos veces, una sin usar y otra usando *pattern matching*).
5. `ambosSon0`: dados dos números racionales, decide si ambos son iguales a 0 (hacerlo dos veces, una sin usar y otra usando *pattern matching*).
6. `esMultiploDe`: dados dos números naturales, decidir si el primero es múltiplo del segundo.
7. `digitoUnidades`: dado un número natural, extrae su dígito de las unidades.
8. `digitoDecenas`: dado un número natural, extrae su dígito de las decenas.

Observación: Cuando el problema en cuestión trata sobre números naturales, se puede simplemente usar el tipo `Int` e ignorar el comportamiento del programa si el usuario decide ejecutarlo usando para los parámetros enteros negativos o 0.