# Práctica 1

## Programación Funcional en Haskell

Primera parte

Paradigmas de Lenguajes de Programación

Departamento de Ciencias de la Computación
Universidad de Buenos Aires

27 de agosto de 2024

## Repaso: usando GHCi

Cómo empezar:

```
$ ghci
Loading ...
Prelude>:q
Leaving GHCi.
$ ghci test.hs
Loading ...
[1 of 1] Compiling Main ( test.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Otros comandos útiles:

- Para recargar: `:r`
- Para cargar otro archivo: `:l archivo.hs`
- Para conocer el tipo de una expresión: `:t True`



## Currificación y aplicación parcial

```
prod :: (Int, Int) -> Int
prod (x, y) = x * y

prod' :: Int -> (Int -> Int)
(prod' x) y = x * y
```

Las funciones en Haskell siempre toman un único argumento.

Entonces ¿qué hacen estas funciones?

- `prod` recibe una tupla de dos elementos.
- `prod'` es una función que **toma un** $x$ de tipo `Int` y **devuelve una función** de tipo `Int -> Int`, cuyo comportamiento es tomar un entero y multiplicarlo por $x$.
  En particular, `(prod' 2)` es la función que duplica.
  Una definición equivalente de `prod'` usando funciones anónimas:
  `prod' x = \y -> x*y`

Decimos que `prod'` es la versión currificada de `prod`.



## curry – uncurry

**Ejercicio**

Definir las siguientes funciones:

1. `curry :: ((a,b) -> c) -> (a -> b -> c)`
   que devuelve la versión currificada de una función no currificada.
2. `uncurry :: (a -> b -> c) -> ((a,b) -> c)`
   que devuelve la versión no currificada de una función currificada.

Los paréntesis en gris no son necesarios, pero es útil escribirlos cuando estamos aprendiendo y queremos ver más explícitamente que estamos devolviendo una función.

```
curry f x y = f (x, y)

uncurry f (x, y) = f x y
```

## Aplicación parcial

### Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int
prod x y = x * y
```

Definimos `doble x = prod 2 x`

1. ¿Cuál es el tipo de doble? `doble :: Int -> Int`
2. ¿Qué pasa si cambiamos la definición por `doble = prod 2`? Sigue siendo la misma función
3. ¿Qué significa `(+) 1`? Es una función que al 1 le suma el siguiente valor ingresado
4. Definir las siguientes funciones de forma similar a `(+)1`:
    - `triple :: Float -> Float`
    - `esMayorDeEdad :: Int -> Bool`

```
triple = (*) 3.0
esMayorDeEdad = (<=) 18
```



## Funciones muy útiles

### Ejercicios

1. Implementar y dar los tipos de las siguientes funciones:
    a. `(.)` que compone dos funciones. Por ejemplo:
       `((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
    b. `flip` que invierte los argumentos de una función. Por ejemplo:
       `flip (\x y -> x - y) 1 5` devuelve 4.
    c. `($)` que aplica una función a un argumento. Por ejemplo:
       `id $ 6` devuelve 6.
    d. `const` que, dado un valor, retorna una función constante que devuelve siempre ese valor. Por ejemplo:
       `const 5 ''casa''` devuelve 5.

   Al final
2. ¿Qué hace `flip ($) 0`? Aplica 0 a la siguiente función
3. ¿Y `(==0) . (flip mod 2)`? Devuelve True si el parámetro ingresado es múltiplo de 2

Pueden ver más funciones útiles en la sección Útil del Campus.



## Listas

Hay varias **macros** para definir listas:

- Por extensión
  Esto es, dar la lista explícita, escribiendo todos sus elementos.
  Por ejemplo: `[4, 3, 3, 4, 6, 5, 4, 5, 4, 5]`.
- Secuencias
  Son progresiones aritméticas en un rango particular.
  Por ejemplo: `[3..7]` es la lista que tiene todos los números enteros entre 3 y 7, mientras que `[2, 5..18]` es la lista que contiene 2, 5, 8, 11, 14 y 17.
- Por comprensión
  Se definen de la siguiente manera:
  `[expresión | selectores, condiciones]`
  Por ejemplo: `[(x,y) | x <- [0..5], y <- [0..3], x+y==4]` es la lista que tiene los pares (1,3), (2,2), (3,1) y (4,0).



## Listas infinitas

```
esPrimo :: Int -> Bool
esPrimo n = all (\x -> n `mod` x /= 0) [2..(n-1)]
```

Haskell también nos permite trabajar con listas infinitas.

La función all devuelve true si todos los elementos de una lista cumplen con la condición

Algunos ejemplos:

- `naturales = [1..]`
  1, 2, 3, 4, ...
- `multiplosDe3 = [0,3..]`
  0, 3, 6, 9, ...
- `repeat ''hola''`
  "hola", "hola", "hola", "hola", ...
- `primos = [n | n <- [2..], esPrimo n]`
  (asumiendo `esPrimo` definida) 2, 3, 5, 7, ...
- `infinitosUnos = 1 : infinitosUnos`
  1, 1, 1, 1, ...

¿Cómo es posible trabajar con listas infinitas sin que se cuelgue?

## Evaluación lazy

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

nUnos :: Int -> [Int]
nUnos n = take n infinitosUnos
```

```
maximo (x:[]) = x
maximo (x:xs) = if x > maximo xs
                then x
                else maximo xs
```

```
minimo (x:[]) = x
minimo (x:xs) = if x < minimo xs
                then x
                else minimo xs
```

- Si ejecutamos `nUnos 2`...
  `nUnos 2` → `take 2 infinitosUnos` → `take 2 (1:infinitosUnos)`→ `1 : take (2-1) infinitosUnos` → `1 : take 1 infinitosUnos` → `1 : take 1 (1:infinitosUnos)`→ `1 : 1: take (1-1) infinitosUnos` → `1 : take 0 infinitosUnos` → `1 : 1 : []`
- ¿Qué sucedería si usáramos otra estrategia de reducción?
- Si para algún término existe una reducción finita, entonces la estrategia de reducción lazy termina.



## Funciones de alto orden

```
listaMasCorta (x:[]) = x
listaMasCorta (x:xs) = if length x < length (listaMasCorta xs)
                       then x
                       else listaMasCorta xs
```

Definamos las siguientes funciones
Precondición: las listas tienen algún elemento.

- `maximo :: Ord a => [a] -> a`
- `minimo :: Ord a => [a] -> a`
- `listaMasCorta :: [[a]] -> [a]`

`listaMasCorta' = mejorSegun (\x y -> length x < length y)`

Siempre hago lo mismo... ¿Se podrá generalizar? ¿Cómo?

**Ejercicio**

- `mejorSegun :: (a -> a -> Bool) -> [a] -> a`
- Reescribir `maximo` y `listaMasCorta` en base a `mejorSegun`

```
mejorSegun f [x] = x
mejorSegun f (x:y:xs) = if f x y
                        then mejorSegun f (x:xs)
                        else mejorSegun f (y:xs)
```

`maximo' = mejorSegun (>)`



## Esquemas de recursión sobre listas: filter

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x rec -> if p x then (:) x rec else rec) []
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) =
    if p x
    then x : filter p xs
    else filter p xs
```

**Ejercicios**

`deLongitudN n = filter (\x -> length x == n)`

- Redefinir `filter` usando `foldr`
- Definir usando `filter`:
  - `deLongitudN :: Int -> [[a]] -> [[a]]`
  - `soloPuntosFijosEnN :: Int -> [Int->Int] -> [Int->Int]`
    Dados un número $n$ y una lista de funciones, deja las funciones que al aplicarlas a $n$ dan $n$

`soloPuntosFijosEnN n = filter (\f -> f n == n)`



## Esquemas de recursión sobre listas: map

```
map' :: (a -> b) -> [a] -> [b]
map' f = foldr (\x rec -> f x : rec) []
```

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

**Ejercicio**

- Redefinir `map` usando `foldr`
- Definir usando `map`: `reverseAnidado = reverse . map reverse`
  - `reverseAnidado :: [[Char]] -> [[Char]]` que, dada una lista de strings, devuelve una lista con cada string dado vuelta y la lista completa dada vuelta. Por ejemplo: `reverseAnidado [``quedate'', ``en'', ``casa'']` devuelve `[``asac", ``ne'', ``etadeuq'']`.
    Ayuda: ya existe la función `reverse` que invierte una lista.
  - `paresCuadrados :: [Int] -> [Int]` que, dada una lista de enteros, devuelve una lista con los cuadrados de los números pares y los impares sin modificar.

`paresCuadrados = map (\x -> if even x then x * x else x)`

Definir una expresión equivalente a las siguiente utilizando `map` y `filter`:

**Ejercicio**

`listaComp f xs p = [f x | x <- xs, p x]`

```
listaComp' :: (a -> b) -> [a] -> (a -> Bool) -> [b]
listaComp' f xs p = map f (filter p xs)
```





Ya conocen `foldr` y `foldl`.
Para situaciones en las cuales no hay un caso base claro (ej: no existe el neutro), tenemos las funciones: `foldr1` y `foldl1`.
Permiten hacer recursión estructural sobre listas sin definir un caso base:

- `foldr1` toma como caso base el último elemento de la lista.
- `foldl1` toma como caso base el primer elemento de la lista.

Para ambas, la lista **no** debe ser vacía, y el tipo del resultado debe ser el de los elementos de la lista.

**Ejercicio**

Definir mejorSegún `:: (a -> a -> Bool) -> [a] -> a` usando `foldr1` o `foldl1`.



```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(.) f g x = f (g x)

flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x

($) :: (a -> b) -> a -> b
f $ x = f x

const :: a -> b -> a
const x _ = x
```

```
mejorSegunFoldr1 :: (a -> a -> Bool) -> [a] -> a
mejorSegunFoldr1 f = foldr1 (\x y -> if f x y then x else y)

mejorSegunFoldl1 :: (a -> a -> Bool) -> [a] -> a
mejorSegunFoldl1 f = foldl1 (\x y -> if f x y then x else y)
```