# PROGRAMACIÓN FUNCIONAL

## Modelo Funcional: Currificación


¿VOS SABÍAS QUE MI MAMÁ ES TRADUCTORA DE FRANCÉS, MANOLITO? YO TAMBIÉN SÉ FRANCÉS; SÉ DECIR "PAPÁ" EN FRANCÉS
¿SÍÍ? ¿CÓMO SE DICE, A VER?
PAPÁ
¡AH, ES FÁCIL! ¡SE DICE IGUAL!
¿FÁCIL? ¿IGUAL? ¡NADA DE ESO, EL ASUNTO ES PENSAR EN FRANCÉS! ¡TRATÁ DE DECIR "PAPÁ" PENSÁNDOLO EN FRANCÉS! ¡DALE! ¿A VER? ¡DALE!
¡ES INÚTIL! ¡JAMÁS PODRÉ HABLAR ESE MALDITO IDIOMA!

# Currificación

- Funciones como valores
- Aplicación: currificación.
- Notación. Ventajas.
- Ejemplos.

# Funciones como valores

- Las funciones son valores, al igual que los números, las tuplas, etc.
  - pueden ser argumento de otras funciones
  - pueden ser resultado de otras funciones
  - pueden almacenarse en estructuras de datos
  - pueden ser estructuras de datos
- Funciones que manipulan funciones
  - Las llamamos “de alto orden”, abusando de esa nomenclatura

# Funciones como valores

- Ejemplo

```
compose (f,g) = h  where h x = f (g x)
sqr x = x*x
twice f = g   where g x = f (f x)
aLaCuarta = compose (sqr,sqr)
aLaOctava = compose (sqr,aLaCuarta)
fs = [ sqr, aLaCuarta, aLaOctava, twice sqr ]
```

aLaCuarta 2 $\rightarrow$ ?

- ¿Será cierto que aLaCuarta = twice sqr?

# Aplicación del alto orden

- Considere las siguientes definiciones

```
suma' :: ??
suma' (x,y) = x+y

suma :: ??
suma x = f   where  f y = x+y
```

- ¿Qué tipo tienen las funciones?
- ¿Qué similitudes observa entre suma y suma'?
- ¿Qué diferencias observa entre ellas?

# Aplicación del alto orden

- Similitudes
  - ambas expresan la suma de dos enteros:
    suma' (x,y) = (suma x) y, para x e y cualesquiera
- Diferencias
  - una toma un par y retorna un número;
    la otra toma un número y retorna una *función*
  - con suma se puede definir la función sucesor
    sin usar variables extra:
    succ = suma 1

# Currificación

- Correspondencia entre cada función de múltiples parámetros y una de alto orden que retorna una función intermedia que completa el trabajo.
  - Por cada f' definida como

```
f' :: (a,b) -> c
f' (x,y) = e
```

  siempre se puede escribir

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

# Currificación

- Correspondencia entre los tipos

  (a,b) -> c  y  a -> (b -> c)

  curry :: ((a,b) -> c) -> (a -> (b -> c))
  curry …

  uncurry :: (a -> (b -> c)) -> ((a,b) -> c)
  uncurry …

  - Se puede demostrar que

    curry (uncurry f) = f
    uncurry (curry f') = f'

# Currificación - Sintaxis

- ¿Cómo escribimos una función currificada y su aplicación?
- Considerar las siguientes definiciones

```
twice :: (Int->Int) -> (Int -> Int)
twice1 f = g where g x = f (f x)
twice2 f = \x -> f (f x)
(twice3 f) x = f (f x)
```

- ¿Son equivalentes? ¿Cuál es preferible? ¿Por qué?

# Currificación

- ¿Qué pasa con un ejemplo más grande?
  - Consideremos una función para sumar 5 números

```
sum5' :: (Int, Int, Int, Int, Int) -> Int
sum5' (x,y,z,v,w) = x+y+z+v+w
```

vs.

```
sum5 :: ??
sum5 = ??
```

# Currificación

- ¿Qué pasa con un ejemplo más grande? (cont.)
  - Con nombres intermedios...

```
sum5 :: Int -> (Int -> (Int -> (Int -> (Int -> Int))))
sum5 x = sum4
        where sum4 y = sum3
          where sum3 z = sum2
            where sum2 v = sum1
              where sum1 w = x+y+z+v+w
```

# Currificación

- ¿Qué pasa con un ejemplo más grande? (cont.)
  - Con aplicación reiterada...

```
sum5 :: Int -> (Int -> (Int -> (Int -> (Int -> Int))))
((((sum5 x) y) z) v) w = x+y+z+v+w
```

vs.

```
sum5' :: (Int, Int, Int, Int, Int) -> Int
sum5' (x,y,z,v,w) = x+y+z+v+w
```

# Currificación

- ¿Cómo podemos evitar usar paréntesis? Convenciones de notación
  - La aplicación de funciones asocia a izquierda
  - El tipo de las funciones asocia a derecha

```
suma :: Int ->Int ->Int
suma x y = x+y
```

```
suma :: Int -> (Int ->Int)
(suma x) y = x+y
```

# Currificación

- ¿Qué pasa con un ejemplo más grande? (cont.)
  - Consideremos una función para sumar 5 números

```
sum5 :: Int -> (Int -> (Int -> (Int -> (Int -> Int))))
((((sum5 x) y) z) v) w = x+y+z+v+w
```

vs.

```
sum5 :: Int -> Int -> Int -> Int -> Int -> Int
sum5 x y z v w = x+y+z+v+w
```

# Currificación

- Por abuso de lenguaje

  ```
  suma :: Int -> Int -> Int
  suma x y = x+y
  ```

  suma es una función que toma dos enteros y retorna otro entero.

en lugar de

```
suma :: Int -> (Int -> Int)
(suma x) y = x+y
```

suma es una función que toma un entero y devuelve una función, la cual toma un entero y devuelve otro entero.

# Currificación

- Ventajas.
  - Mayor expresividad

```
derive :: (Float -> Float) -> (Float -> Float)
derive f x = (f (x+h) - f x) / h   where h = 0.0001
```

  - Aplicación parcial

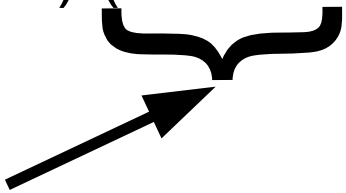
```
derive f          ( = \x -> (f (x+h) - f x) / h )
```

  - Modularidad para tratamiento de código
    - Al inferir tipos
    - Al transformar programas

# Aplicación Parcial

- Definir un función que calcule la derivada n-ésima de una función

```
deriveN :: Int -> (Float -> Float) -> (Float -> Float)
deriveN 0 f = f
deriveN n f = deriveN (n-1) (derive f)
```



Aplicación parcial de derive.

- ¿Cómo lo haría con derive'?

# Expresividad

- Definir un función que calcule la aplicación n veces de una función

```
many :: Int -> (a -> a) -> (a -> a)
many 0 f x = x
many n f x = many (n-1) f (f x)
```

- Se pueden probar (o definir) muchas ideas ya vistas...

```
twice = many 2
deriveN n = many n derive
```

# Currificación

- Decir que algo está currificado es una CUESTIÓN DE INTERPRETACIÓN

```
movePoint :: (Int, Int) -> (Int, Int)
movePoint (x,y) = (x+1,y+1)

distance :: (Int, Int) -> Int
distance (x,y) = sqrt (sqr x + sqr y)
```

  - ¿Están currificadas? ¿Por qué?

# Resumen

- Currificación.
- Ventajas: aumento de expresividad.