Paradigmas de Programación

Fundamentos de programación funcional

2do cuatrimestre de 2025

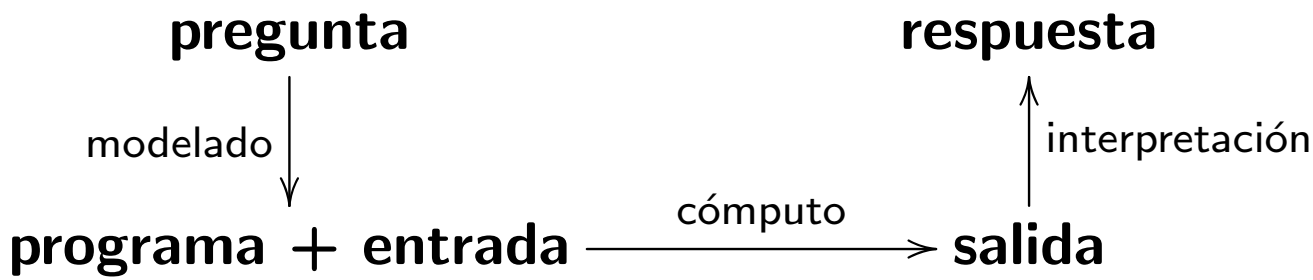
Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Fundamentos de programación funcional

Ejercicios

Programación funcional

Un problema central en computación es el de procesar información:



La programación funcional consiste en definir funciones y aplicarlas para procesar información.

Las "funciones" son verdaderamente funciones (parciales):

- Aplicar una función no tiene efectos secundarios.
- A una misma entrada corresponde siempre la misma salida.
- Las estructuras de datos son inmutables.

Las funciones son datos como cualquier otro:

- Se pueden pasar como parámetros.
- Se pueden devolver como resultados.
- Pueden formar parte de estructuras de datos.
  (Ej. árbol binario en cuyos nodos hay funciones).

Programación funcional

Un programa funcional está dado por un conjunto de ecuaciones:

Ejemplo

```
longitud []       = 0
longitud (x : xs) = 1 + longitud xs
```

```
    longitud [10, 20, 30]
≡   longitud (10 : (20 : (30 : [])))
=   1 + longitud (20 : (30 : []))
=   1 + (1 + (longitud (30 : [])))
=   1 + (1 + (1 + longitud []))
=   1 + (1 + (1 + 0))
=   1 + (1 + 1)
=   1 + 2
=   3
```

Expresiones

Las **expresiones** son secuencias de símbolos que sirven para representar datos, funciones y funciones aplicadas a los datos. (Recordemos: las funciones también son datos).

Una expresión puede ser:

1. Un **constructor**:

   ```
   True  False  []  (:)  0  1  2  ...
   ```

2. Una **variable**:

   ```
   longitud  ordenar  x  xs  (+)  (*)  ...
   ```

3. La **aplicación** de una expresión a otra:

   ```
   ordenar lista
   not True
   not (not True)
   (+) 1
   ((+) 1) (alCuadrado 5)
   ```

4. También hay expresiones de otras formas, como veremos. Las tres de arriba son las fundamentales.

Expresiones

Convenimos en que la aplicación es **asociativa a izquierda**:

```
f x y       ≡  (f x) y          ≢  f (x y)
f a b c d   ≡  (((f a) b) c) d
```

Ejemplo

```
    [1, 2]
≡   1 : [2]
≡   (:) 1 [2]
≡   ((:) 1) [2]
≡   ((:) 1) (2 : [])
≡   ((:) 1) ((:) 2 [])
≡   ((:) 1) (((:) 2) [])
```

Expresiones

Ejemplo

```
sumarUno = (+) 1
```

```
                  sumarUno (sumarUno 5)
              =   ((+) 1) (sumarUno 5)
              ≡   1 + sumarUno 5
              =   1 + ((+) 1) 5
              ≡   1 + (1 + 5)
              =   1 + 6
              =   7
```

Tipos

Hay secuencias de símbolos que no son expresiones bien formadas.

Ejemplo

```
1,,2      )f x(
```

Hay expresiones que están bien formadas pero no tienen sentido.

Ejemplo

```
True + 1
0 1
[[], (+)]
```

Tipos

Un **tipo** es una especificación del invariante de un dato o de una función.

Ejemplo

```
99          :: Int
not         :: Bool -> Bool
not True    :: Bool
(+)         :: Int -> (Int -> Int)
(+) 1       :: Int -> Int
((+) 1) 2   :: Int
```

El tipo de una función expresa un **contrato**.

Condiciones de tipado

Para que un programa esté **bien tipado**:

1. Todas las expresiones deben tener tipo.
2. Cada variable se debe usar siempre con un mismo tipo.
3. Los dos lados de una ecuación deben tener el mismo tipo.
4. El argumento de una función debe tener el tipo del dominio.
5. El resultado de una función debe tener el tipo del codominio.

$$\frac{\texttt{f :: a -> b} \qquad \texttt{x :: a}}{\texttt{f x :: b}}$$

Sólo tienen sentido los programas bien tipados.

No es necesario escribir explícitamente los tipos. (Inferencia).

Tipos

Convenimos en que "->" es **asociativo a derecha**:

```
a -> b -> c       ≡  a -> (b -> c)   ≢  (a -> b) -> c
a -> b -> c -> d  ≡  a -> (b -> (c -> d))
```

Ejemplo

```
suma4 :: Int -> Int -> Int -> Int -> Int
suma4 a b c d = a + b + c + d
```

Se puede pensar así:

```
suma4 :: Int -> (Int -> (Int -> (Int -> Int)))
(((suma4 a) b) c) d = a + b + c + d
```

Polimorfismo

Hay expresiones que tienen más de un tipo.
Usamos *variables de tipo* `a`, `b`, `c` para denotar tipos desconocidos:

```
id   ::  a -> a
[]   ::  [a]
(:)  ::  a -> [a] -> [a]
fst  ::  (a, b) -> a
snd  ::  (a, b) -> b
```

Ejemplo

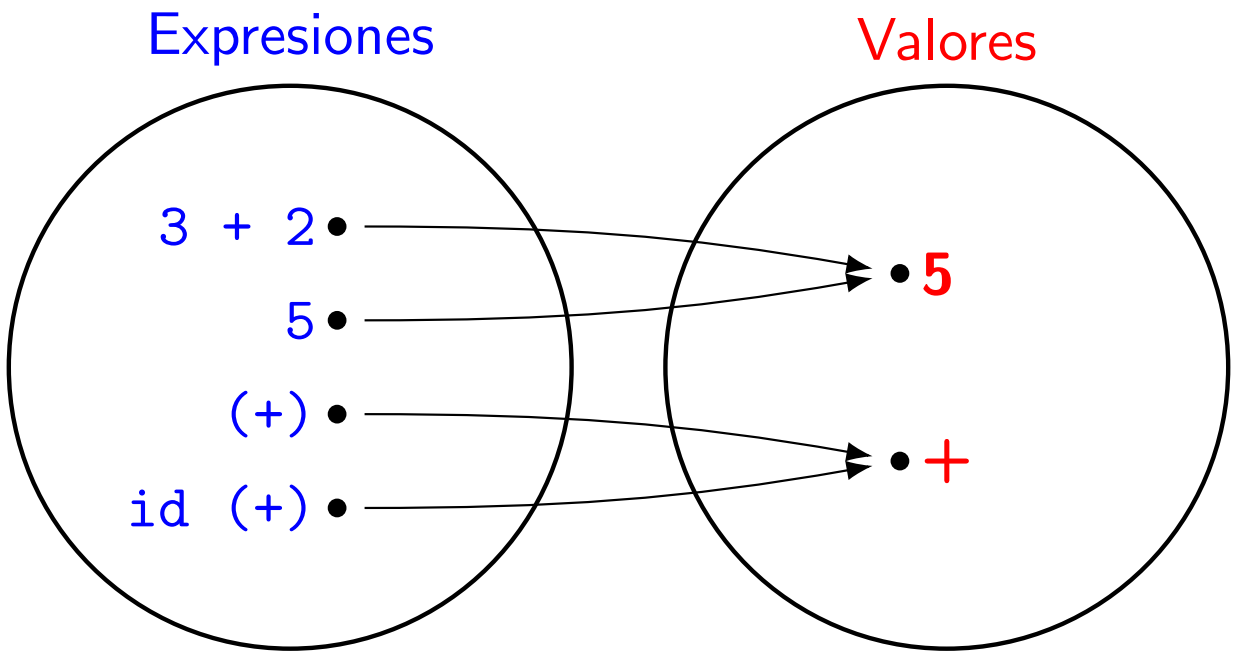
```
flip f x y = f y x
```

¿Qué tipo tiene `flip`?

```
    flip (:) [2, 3] 1
=   (:) 1 [2, 3]
≡   1 : [2, 3]
=   [1, 2, 3]
```

Modelo de cómputo

Dada una expresión, se computa su *valor* usando las ecuaciones:



Hay expresiones bien tipadas que no tienen valor. Ej.: `1 / 0`.
Decimos que dichas expresiones se indefinen o que tienen valor $\bot$.

Modelo de cómputo

Un programa funcional está dado por un conjunto de ecuaciones.
Más precisamente, por un conjunto de **ecuaciones orientadas**.

Una ecuación `e1 = e2` se interpreta desde dos puntos de vista:

1. **Punto de vista denotacional.**
   Declara que `e1` y `e2` tienen el mismo significado.
2. **Punto de vista operacional.**
   Computar el valor de `e1` se reduce a computar el valor de `e2`.

Modelo de cómputo

El lado *izquierdo* de una ecuación no es una expresión arbitraria. Debe ser una función aplicada a **patrones**.

Un patrón puede ser:

1. Una variable.
2. Un comodín _.
3. Un constructor aplicado a patrones.

El lado izquierdo no debe contener variables repetidas.

Ejemplo

¿Cuáles ecuaciones están sintácticamente bien formadas?

```
sumaPrimeros (x : y : z : _) = x + y + z
predecesor (n + 1) = n
iguales x x = True
```

Modelo de cómputo

Evaluar una expresión consiste en:

1. Buscar la subexpresión más externa que coincida con el lado izquierdo de una ecuación.
2. Reemplazar la subexpresión que coincide con el lado izquierdo de la ecuación por la expresión correspondiente al lado derecho.
3. Continuar evaluando la expresión resultante.

La evaluación se detiene cuando se da uno de los siguientes casos:

1. La expresión es un constructor o un constructor aplicado.

   `True` `(:) 1` `[1, 2, 3]`

2. La expresión es una función *parcialmente* aplicada.

   `(+)` `(+) 5`

3. Se alcanza un *estado de error*.
   Un estado de error es una expresión que no coincide con las ecuaciones que definen a la función aplicada.

Modelo de cómputo

Ejemplo: resultado — constructor

```
tail :: [a] -> [a]
tail (_ : xs) = xs
```

```
tail (tail [1, 2, 3]) ⇝ tail [2, 3] ⇝ [3]
```

Ejemplo: resultado — función parcialmente aplicada

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

```
const (const 1) 2  ⇝  const 1
```

Modelo de cómputo

Ejemplo: indefinición — error

```
head :: [a] -> a
head (x : _) = x
```

```
head (head [[], [1], [1, 1]]) ⇝ head []
                              ⇝ ⊥
```

Ejemplo: indefinición — no terminación

```
loop :: Int -> a
loop n = loop (n + 1)
```

```
loop 0 ⇝ loop (1 + 0)
       ⇝ loop (1 + (1 + 0))
       ⇝ loop (1 + (1 + (1 + 0)))
       ...
```

Modelo de cómputo

Ejemplo: evaluación no estricta

```
indefinido :: Int
indefinido = indefinido
```

```
        head (tail [indefinido, 1, indefinido])
⇝       head [1, indefinido]
⇝       1
```

Modelo de cómputo

Ejemplo: listas infinitas

```
desde :: Int -> [Int]
desde n = n : desde (n + 1)
```

```
    desde 0
⇝   0 : desde 1
⇝   0 : (1 : desde 2)
⇝   0 : (1 : (2 : desde 3))   ⇝   ...
```

```
    head (tail (desde 0))
⇝   head (tail (0 : desde 1))
⇝   head (desde 1)
⇝   head (1 : desde 2)
⇝   1
```

Modelo de cómputo

Nota. En Haskell, el orden de las ecuaciones es relevante.
Si hay varias ecuaciones que coinciden se usa siempre la primera.

Ejemplo

```
esCorta (_ : _ : _) = False
esCorta _           = True
```

```
esCorta []      ⇝  True
esCorta [1]     ⇝  True
esCorta [1, 2]  ⇝  False
```

Funciones de orden superior

Definamos la composición de funciones ("`g . f`").

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

Otra forma de definirla (usando la notación "lambda"):

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

Funciones de orden superior

¿Qué tienen en común las siguientes funciones?

```
dobleL :: [Float] -> [Float]
dobleL []       = []
dobleL (x : xs) = x * 2 : dobleL xs

esParL :: [Int] -> [Bool]
esParL []       = []
esParL (x : xs) = x `mod` 2 == 0 : esParL xs

longitudL :: [[a]] -> [Int]
longitudL []       = []
longitudL (x : xs) = length x : longitudL xs
```

Todas ellas siguen el esquema:

```
g []       = []
g (x : xs) = f x : g xs
```

Funciones de orden superior

¿Cómo se puede abstraer el esquema?

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

dobleL    xs = map (\ x -> x * 2) xs
esParL    xs = map (\ x -> x `mod` 2 == 0) xs
longitudL xs = map length xs
```

Otra manera:

```
dobleL    = map (* 2)
esParL    = map ((== 0) . (`mod` 2))
longitudL = map length
```

Funciones de orden superior

¿Qué relación hay entre las siguientes funciones?

```
negativos :: [Int] -> [Int]
negativos []       = []
negativos (x : xs) = if x < 0
                     then x : negativos xs
                     else negativos xs

noVacias :: [[a]] -> [[a]]
noVacias []       = []
noVacias (x : xs) = if not (null x)
                    then x : noVacias xs
                    else noVacias xs
```

Ambas siguen el esquema:

```
g []       = []
g (x : xs) = if p x
             then x : g xs
             else g xs
```

Funciones de orden superior

¿Cómo se puede abstraer el esquema?

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

negativos = filter (< 0)
noVacias  = filter (not . null)
```

Ejercicio (tarea)

```
merge     :: (a -> a -> Bool) -> [a] -> [a] -> [a]
mergesort :: (a -> a -> Bool) -> [a] -> [a]
```

El primer parámetro es una función que determina una relación de orden total entre los elementos de tipo `a`.

Ejercicios

Algunos esquemas de recursión

a) Definir una función:

```
operatoria :: (a -> a -> a) -> [a] -> a
```

que dada una operación binaria (que asumimos asociativa) y una lista no vacía devuelve el resultado de hacer la operación entre todos los elementos.

Informalmente:

$$\texttt{operatoria}\ (\bigstar)\ [x_1, x_2, \ldots, x_n] = x_1 \bigstar x_2 \bigstar \ldots \bigstar x_n.$$

b) Definir la sumatoria y la productoria como casos particulares.

Algunos esquemas de recursión

a) Definir una función:

```
mientras :: (a -> Bool) -> (a -> a) -> a -> a
```

que dada una condición, una función de transformación de estados y un estado inicial devuelva el resultado final que se alcanza aplicando repetidamente la función de transformación mientras se cumpla la condición.

b) Usando la función `mientras`, definir la función que computa el n-ésimo elemento de la sucesión de Fibonacci de manera iterativa.

Recordemos que:

$$F_0 = 0 \qquad F_1 = 1 \qquad F_{n+2} = F_n + F_{n+1}$$

Árboles binarios infinitos

Un **árbol binario** (AB) se representa como un valor del tipo:

```
data AB a = Nil | Bin (AB a) a (AB a)
```

Un **AB infinito** (ABI) se representa como un valor del tipo:

```
data ABI a = IBin (ABI a) a (ABI a)
```

Definir una función:

```
podadoDesdeElNivel :: Int -> ABI a -> AB a
```

de tal modo que `podadoDesdeElNivel` *n a* denota el AB que resulta de podar el árbol *a* a partir del nivel *n*.

Árboles binarios infinitos

Consideremos los tipos de datos de las **direcciones**, y los **caminos** (listas finitas de direcciones):

```
data Dirección = Izq | Der
type Camino = [Dirección]
```

Un árbol binario infinito cuyos nodos tienen datos de tipo a se puede representar también como una **función de caminos** que, dado un camino, indica el valor del nodo en dicha posición:

```
type FunciónDeCaminos a = Camino -> a
```

Ejercicio

a) Definir una función:

```
funciónDeCaminosDe :: ABI a -> FunciónDeCaminos a
```

que dado un ABI devuelve su función de caminos.

b) Definir la función inversa:

```
abiDe :: FunciónDeCaminos a -> ABI a
```

que dada una función de caminos devuelve su ABI.

¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ? ? ? ? ? ? ? ? ?

Lectura recomendada

Capítulo 4 del libro de Bird.

Richard Bird. *Thinking functionally with Haskell*.
Cambridge University Press, 2015.

Comentarios: tipos

Ojo. Dijimos:

> "Cada variable se debe usar siempre con un mismo tipo."

¿Está bien tipado el siguiente programa?

```
sucesor :: Int -> Int
sucesor x = x + 1

opuesto :: Bool -> Bool
opuesto x = not x
```

Sí. Hay dos "x" con distinto tipo pero son variables distintas.
El programa se podría reescribir así:

```
sucesor x = x + 1
opuesto y = not y
```