

Paradigmas de Programación

Introducción a la materia

1er cuatrimestre de 2024

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Presentación de la materia

Introducción a la programación funcional

Profesores

- ▶ Pablo Barenbaum
- ▶ Alejandro Díaz-Caro

Jefes de trabajos prácticos

- ▶ Christian Cossio Mercado
- ▶ Daniela Marottoli
- ▶ Gabriela Steren

Ayudantes de primera

- ▶ Gonzalo Pablo Fernández
("Perla")
- ▶ Malena Ivnisky
- ▶ Edgardo Zoppi

Ayudantes de segunda

- ▶ Lucas Di Salvo
- ▶ Sebastián Felgueras
- ▶ Fernando Frassia
- ▶ Damián Huaier
- ▶ Juan Pablo Miceli
- ▶ Rafael Romani
- ▶ Daniel Wappner

Días y horarios de cursada

- ▶ Martes de 17:00 a 22:00
- ▶ Viernes de 17:00 a 22:00

generalmente práctica
generalmente teórica

Modalidad de evaluación

Parciales

- ▶ Primer parcial
- ▶ Segundo parcial
- ▶ Recuperatorio del primer parcial
- ▶ Recuperatorio del segundo parcial

martes 21 de mayo

viernes 5 de julio

martes 16 de julio

martes 23 de julio

Trabajos prácticos

- ▶ TP 1 (con su recuperatorio)
- ▶ TP 2 (con su recuperatorio)

Los TPs son en **grupos de 4 integrantes**.

Examen final

(La materia no es promocionable).

Vías de comunicación

Docentes → alumnxs

Avisos a través del campus.

Alumnxs → docentes

Lista de correo: `plp-docentes@dc.uba.ar`

(para consultas administrativas)

Discusión entre estudiantes fuera del horario de la materia

Servidor de Discord: `https://tinyurl.com/plpdiscord`

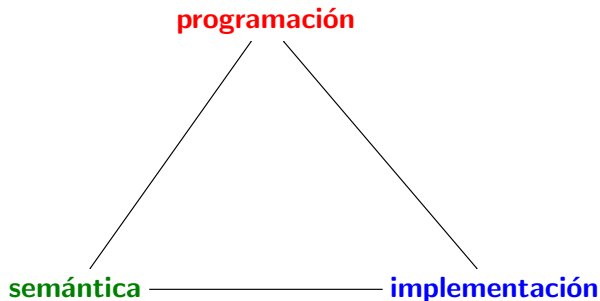
(con eventual participación de docentes)

Página web de la materia

`https://campus.exactas.uba.ar/course/view.php?id=4170`

Contenidos

Tres aspectos de los **lenguajes de programación**:



Cronograma

Programación funcional	1,5 semanas
Razonamiento ecuacional	1 semana
Lógica proposicional	1 semana
Cálculo- λ	2 semanas
Interpretación y compilación	2 semanas
(Repaso)	
Primer parcial	
Lógica de primer orden	1 semana
Resolución	1 semana
Programación lógica	1,5 semanas
Programación orientada a objetos	1 semana
(Repaso)	
Segundo parcial	

Motivación: **programación**

Los lenguajes de programación tienen distintas **características**.

- ▶ Etiquetado dinámico vs. tipado estático.
- ▶ Administración manual vs. automática de memoria.
- ▶ Funciones de primer orden vs. funciones de orden superior.
- ▶ Mutabilidad vs. inmutabilidad.
- ▶ Alcance dinámico vs. estático.
- ▶ Resolución de nombres temprana vs. tardía.
- ▶ Inferencia de tipos.
- ▶ Determinismo vs. no determinismo.
- ▶ Pasaje de parámetros por copia o por referencia.
- ▶ Evaluación estricta (por valor) o diferida (por necesidad).
- ▶ Tipos de datos inductivos, co-inductivos, GADTs, familias dependientes.
- ▶ *Pattern matching*, unificación.
- ▶ Polimorfismo paramétrico.
- ▶ Subclasificación, polimorfismo de subtipos, herencia simple vs. múltiple.
- ▶ Estructuras de control no local.
- ▶ ...

Motivación: programación

Distintas características permiten abordar un mismo problema de distintas maneras.

$$\begin{array}{r} \text{CMXXIV} \\ + \text{MCXLI} \\ \hline \end{array}$$

$$\begin{array}{r} 924 \\ + 1141 \\ \hline \end{array}$$

$$C = \{(x, y) \mid x^2 + y^2 = r^2\} \quad C = \{(r \sin \theta, r \cos \theta) \mid 0 \leq \theta < 2\pi\}$$

```
r := 1
while n > 0 {
  r := r * n
  n := n - 1
}
```

```
foldl (*) 1 [1..n]
```

Motivación: **semántica**

Dependemos del *software* en aplicaciones críticas.

- ▶ Telecomunicaciones.
- ▶ Procesos industriales.
- ▶ Reactores nucleares.
- ▶ Equipamiento médico.
- ▶ Previsión meteorológica.
- ▶ Aeronáutica.
- ▶ Vehículos autónomos.
- ▶ Transacciones monetarias.
- ▶ Análisis de datos en ciencia o toma de decisiones.
- ▶ ...

Las fallas cuestan recursos monetarios y vidas humanas.

Motivación: **semántica**

- ¿Podemos confiar en que un programa hace lo que queremos?
- ¿Y si el programa está escrito por el enemigo?
- ¿Y si el programa está escrito por un gran modelo de lenguaje?

Objetivo

- ▶ Probar teoremas sobre el comportamiento de los programas.
- ▶ ¿Cómo darle significado matemático a los programas?
- ▶ En AED vimos una manera de hacerlo (triplas de Hoare).
- ▶ En PLP veremos otras maneras de dar semántica.

Motivación: **implementación**

Una computadora física ejecuta programas escritos en un lenguaje.
(El “código máquina”).

¿Cómo es capaz de ejecutar programas escritos en otros lenguajes?

- ▶ Interpretación (o evaluación).
- ▶ Chequeo e inferencia de tipos.
- ▶ Compilación (traducción de un lenguaje a otro).

Bibliografía (no exhaustiva)

Lógica proposicional y de primer orden

Logic and Structure

D. van Dalen.

Semántica y fundamentos de la implementación

Introduction to the Theory of Programming Languages

J.-J. Lévy, G. Dowek. Springer, 2010.

Types and Programming Languages

B. Pierce. The MIT Press, 2002.

Programación funcional

Introduction to Functional Programming using Haskell

R. Bird. Prentice Hall, 1998.

Programación lógica

Logic Programming with Prolog

M. Bramer. Springer-Verlag, 2013.

Programación orientada a objetos

Smalltalk-80 the Language and its Implementation

A. Goldberg, D. Robson. Addison-Wesley, 1983.

Presentación de la materia

Introducción a la programación funcional

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]`
 \equiv `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**:

$$\begin{aligned} f \ x \ y &\equiv (f \ x) \ y && \not\equiv f \ (x \ y) \\ f \ a \ b \ c \ d &\equiv (((f \ a) \ b) \ c) \ d \end{aligned}$$

Ejemplo

$$\begin{aligned} &[1, 2] \\ \equiv &1 : [2] \\ \equiv &(:) 1 [2] \\ \equiv &((:)) 1 [2] \\ \equiv &((:)) 1 (2 : []) \\ \equiv &((:)) 1 ((:)) 2 [] \\ \equiv &((:)) 1 (((:)) 2 []) \end{aligned}$$

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.

$$\frac{f :: a \rightarrow b \quad x :: a}{f \ x :: b}$$

Sólo tienen sentido los programas bien tipados.

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

Tipos

Convenimos en que “ \rightarrow ” es **asociativo a derecha**:

$$\begin{aligned} a \rightarrow b \rightarrow c &\equiv a \rightarrow (b \rightarrow c) \quad \text{✗} \quad (a \rightarrow b) \rightarrow c \\ a \rightarrow b \rightarrow c \rightarrow d &\equiv a \rightarrow (b \rightarrow (c \rightarrow d)) \end{aligned}$$

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

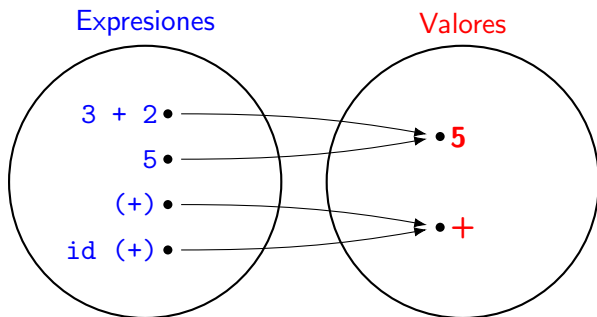
```
flip f x y = f y x
```

¿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 \perp .

Modelo de cómputo

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

Una ecuación $e_1 = e_2$ se interpreta desde dos puntos de vista:

1. **Punto de vista denotacional.**

Declara que e_1 y e_2 tienen el mismo significado.

2. **Punto de vista operacional.**

Computar el valor de e_1 se reduce a computar el valor de e_2 .

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
```

$$\text{tail (tail [1, 2, 3])} \rightsquigarrow \text{tail [2, 3]} \rightsquigarrow [3]$$

Ejemplo: resultado — función parcialmente aplicada

```
const :: a -> b -> a
```

```
const x y = x
```

$$\text{const (const 1) 2} \rightsquigarrow \text{const 1}$$

Modelo de cómputo

Ejemplo: indefinición — error

`head :: [a] -> a`

`head (x : _) = x`

`head (head [], [1], [1, 1])` \rightsquigarrow `head []`
 \rightsquigarrow \perp

Ejemplo: indefinición — no terminación

`loop :: Int -> a`

`loop n = loop (n + 1)`

`loop 0` \rightsquigarrow `loop (1 + 0)`
 \rightsquigarrow `loop (1 + (1 + 0))`
 \rightsquigarrow `loop (1 + (1 + (1 + 0)))`
 \dots

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”).

$$(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$
$$(g \cdot f) \ x = g \ (f \ x)$$

Otra forma de definirla (usando la notación “lambda”):

$$(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$
$$g \cdot f = \lambda \ x \rightarrow g \ (f \ x)$$

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?

```
suma :: Int -> Int -> Int
```

```
suma x y = x + y
```

```
suma' :: (Int, Int) -> Int
```

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

Están relacionadas del siguiente modo:

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

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

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

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

Dentro de algunas clases, veremos que se puede demostrar:

```
suma = curry suma'      suma' = uncurry suma
```

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

```
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`.

i i i i i i i i i i ? ? ? ? ? ? ? ?

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
```