FACULTAD DE INGENIERIA
Universidad de Buenos Aires

75.14 / 95.48 Lenguajes Formales
Prof. Dr. Diego Corsi
Lic. Pablo Bergman

# Programación Funcional

## con ejemplos en APL, FP de John Backus y Clojure

# Paradigmas de Programación

- **Imperativos (énfasis en la ejecución de instrucciones)**
  - Programación Procedimental (p. ej. *Pascal*)
  - Programación Orientada a Objetos (p. ej. *Smalltalk*)

- **Declarativos (énfasis en la evaluación de expresiones)**
  - **Programación Funcional** (p. ej. *Haskell*)
  - Programación Lógica (p. ej. *Prolog*)

**Los lenguajes más utilizados son, en su mayoría, *multiparadigma*:**

**Cabe a los programadores usar el estilo de programación y las construcciones de lenguaje más adecuados para un trabajo determinado.**

# Características de la Programación Funcional

1. **PROGRAMAS COMO FUNCIONES**
2. **FUNCIONES PURAS**
3. **DATOS INMUTABLES**
4. **FUNCIONES DE PRIMERA CLASE**
5. **FUNCIONES DE ORDEN SUPERIOR**
6. **COMPOSICIÓN DE FUNCIONES**
7. **RECURSIVIDAD**

# Características de la Programación Funcional

1

# PROGRAMAS COMO FUNCIONES

- **Estructura de un programa:** un programa consiste de una lista de definiciones de funciones.

- **Ejecución de un programa:** consiste en aplicar las funciones a sus argumentos (según las bases establecidas por el **Cálculo Lambda**).

# Programas como funciones

**Ejemplo: Cálculo del promedio de una colección de números**

**APL:**

```
prom ← {(+/ω) ÷ ρω}
prom 1 3 5 7
4
```

**FP:**

```
Def prom ≡ ÷ ∘ [/+, length]
prom : <1, 3, 5, 7>
4
```

**Clojure:**

```
(defn prom [v] (/ (reduce + v) (count v)))
(prom [1 3 5 7])
4
```

# Características de la Programación Funcional

2

## FUNCIONES PURAS

- **Determinismo:** Ante los mismos argumentos, una función pura siempre retorna el mismo valor. Esto conlleva la **transparencia referencial** (una función pura puede reemplazarse por su valor de retorno).
- **Ausencia de efectos colaterales:** Además del valor de retorno, una función pura no tiene ningún efecto visible sobre el ambiente desde el cual se la invoca.

→ Consecuencia práctica: Se hacen más fáciles las **Pruebas Unitarias**

# Funciones puras

**Ejemplo: función sucesor ( *succ* )**

**APL:**

```
succ ← {ω + 1}
succ 4
5
```

```
uno ← 1
succ ← {ω + uno}
```

función impura
(no determinista)

**FP:**

```
Def succ ≡ + ∘ [id, 1̄]
succ : 4
5
```

**Clojure:**

```
(defn succ [n] (+ n 1))
(succ 4)
5
```

función impura
(efecto colateral)

```
(defn succ [n] (do (def x 1)(+ n x)))
```

# Características de la Programación Funcional

## 3

# DATOS INMUTABLES

- **Las funciones no modifican los datos recibidos como argumentos:** los valores retornados siempre están alojados en otras ubicaciones de memoria (las estructuras de datos son **persistentes**).

  → Consecuencia práctica: Se hace más fácil la **Programación Concurrente**

# Datos inmutables

**Ejemplo: función invertir ( *reverse* )**

**APL:**

```
      L ← 1 2 3 4
      ⌽ L
4 3 2 1
      L
1 2 3 4
```

copia modificada

**Clojure:**

```
(def L '(1 2 3 4))
(reverse L)
(4 3 2 1)
L
(1 2 3 4)
```

copia modificada

# Características de la Programación Funcional

4

# FUNCIONES DE PRIMERA CLASE

- **Soportan las operaciones posibles para las otras entidades del lenguaje:** pueden ser asignadas a una variable, ser pasadas como argumento y ser retornadas por una función.

# Funciones de primera clase

**APL:**

```
f ← ÷
1 f 2
0.5
```

La función ÷ es asignada

**FP:**

```
(> → 1; 2)
(> → 1; 2) : <3, 4>
4
(> → 1; 2) : <4, 3>
4
```

La función 2 es retornada

La función 1 es retornada

**Clojure:**

```
(defn operar [f v] (reduce f v))
(operar + [1 3 5 7])
16
(operar * [1 3 5 7])
105
```

La función + es pasada como argumento

La función * es pasada como argumento

# Características de la Programación Funcional

5

# FUNCIONES DE ORDEN SUPERIOR

Tienen alguna de las siguientes capacidades (o ambas):

- **Recibir funciones como argumento**
- **Retornar una función**

# Funciones de orden superior

**Ejemplo: función reducir ( / o *reduce* )**

**APL:**

```
+/ 1 2 3 4
10
```

**FP:**

```
/+ : <1, 2, 3, 4>
10
```

**Clojure:**

```
(reduce + [1 2 3 4])
10
```

# Funciones de orden superior

**Ejemplo: función aplicar a todos ( α o *map* )**

**APL:**

```
cuad ← {ω × ω}
cuad 1 2 3 4 5
1 4 9 16 25
```

Por defecto, en APL las funciones se aplican a todos

**FP:**

```
Def cuad ≡ × ∘ [id, id]
 α cuad  : <1, 2, 3, 4, 5>
<1, 4, 9, 16, 25>
```

**Clojure:**

```
(defn cuad [n] (* n n))
(map cuad '(1 2 3 4 5))
(1 4 9 16 25)
```

# Funciones de orden superior

**Ejemplo: función aplicar parcialmente ( *partial* )**

**recibe una función de aridad 2 y uno de los argumentos**

**Clojure:**

```
(def hs2sec (partial * 3600))
(hs2sec 2)
7200
```

**retorna otra función de aridad 1**

# Características de la Programación Funcional

6

# COMPOSICIÓN DE FUNCIONES

Consiste en combinar  funciones para formar otra:

- **El valor resultante se calcula aplicando una función a los argumentos. Luego se aplica otra al resultado, y así sucesivamente.**

# Composición de funciones

**Ejemplo: sumatoria de los recíprocos de los números de una colección**

**APL:**

```
sumrecipr ← {+/ ÷ (ω≠0)/ω}
sumrecipr ¯1 0 2 1
0.5
```

**FP:**

```
Def aux ≡ eq ∘ [1, 0̄] → 2; apndl
Def filtrar ≡ (/aux) ∘ apndr ∘ [id, ∅̄]
Def reciproco ≡ ÷ ∘ [1̄, id]
Def mapear ≡ α reciproco
Def reducir ≡ /+
Def sumrecipr ≡ reducir ∘ mapear ∘ filtrar
sumrecipr : <-1, 0, 2, 1>
0.5
```

**Clojure:**

```
(defn sumrecipr [v] (reduce + (map / (filter #(not= % 0) v))))
(sumrecipr [-1 0 2 1])
1/2
```

# Composición de funciones

**Clojure permite componer funciones de diversas maneras:**

## Usando la sintaxis tradicional de LISP

```
(defn sumrecipr [v] (reduce + (map / (filter #(not= % 0) v))))
```

## Usando *comp* y funciones previamente definidas

```
(defn reducir [v] (reduce + v))
(defn mapear [v] (map / v))
(defn filtrar [v] (filter #(not= % 0) v))
(def sumrecipr (comp reducir mapear filtrar))
```

## Usando *comp* y *partial*

```
(def sumrecipr
  (comp (partial reduce +) (partial map /) (partial filter #(not= % 0))))
```

## Usando macros ( ->, ->>, as-> )

```
(defn sumrecipr [v] (->> v (filter #(not= % 0)) (map /) (reduce +)))

(sumrecipr [-1 0 2 1])
1/2
```

# Características de la Programación Funcional

7

## RECURSIVIDAD

**La inexistencia de asignaciones de variables y la falta de construcciones estructuradas como la secuencia y la iteración hacen que las repeticiones de instrucciones se implementen mediante funciones de orden superior o mediante funciones recursivas:**

- **Una función recursiva es aquella que contiene, en el bloque de instrucciones que la definen, una llamada a la propia función, permitiendo que una operación se realice una y otra vez, hasta alcanzar el caso base.**

# Recursividad

**Ejemplo: función factorial ( *fact* o *!* )**

**APL:**

```
fact ← {ω=0: 1 ◊ ω × fact ω-1}
fact 5
120
```

llamada recursiva

```
{ω=0: 1 ◊ ω × ∇ ω-1} 5
120
```

llamada recursiva

**FP:**

```
Def ! ≡ (eq ∘ [id,0̄]) → 1̄; × ∘ [id, ! ∘ - ∘ [id,1̄]]
! : 5
120
```

llamada recursiva

**Clojure:**

```
(defn fact [n] (if (zero? n) 1 (* n (fact (dec n)))))
(fact 5)
120
```

llamada recursiva

# Recursividad

**Clojure permite utilizar recursividad con pila y sin pila:**

```
(defn fact [n] (if (zero? n) 1 (* n (fact (dec n)))))
(fact 5)
120
```

Requiere usar la pila

```
(defn fact [n] (if (zero? n) 1 (* n (recur (dec n)))))
UnsupportedOperationException: Can only recur from tail position
```

```
(defn fact
   ([n] (fact n 1))
   ([n resultado] (if (zero? n) resultado
                      (recur (dec n) (* resultado n)))))
(fact 5)
120
```

No requiere usar la pila

## (recur *expr**)

Note that `recur` is the only non-stack-consuming looping construct in Clojure. There is no tail-call optimization and the use of self-calls for looping of unknown bounds is discouraged. `recur` is functional and its use in tail-position is verified by the compiler.

Fuente: clojure.org