Programación Funcional 1

FIUBA - Técnicas de Diseño

Un ejemplo

```
Sea x = a + b donde
        a = f(1)
        b = f(2)
¿Cual es el valor de x?
```

```
Sea x = a + b donde
        b = f(2)
        a = f(1)
¿Cual es el valor de x?
```

Desconocemos f : $\mathbb{N} \rightarrow \mathbb{N}$

Estos dos problemas matemáticos, ¿tienen la misma solución?
La única diferencia es el orden de las dos definiciones indicadas

Un ejemplo

```
int main () {
        int a = f(1)
        int b = f(2)
        return a + b
}
```

```
int main () {
        int b = f(2)
        int a = f(1)
        return a + b
}
```

Desconocemos f : int → int

Estos dos programas, ¿retornan el mismo número?
La única diferencia es el orden de las dos líneas indicadas

Un ejemplo

```
Sea x = a + b donde
          a = f(1)
          b = f(2)
¿Cuál es el valor de x?
```

```
int main () {
        int b = f(1)
        int a = f(2)
        return a + b
}
```

- En programación, f puede tener un efecto **implícito, no-local**
- En matemática eso es imposible

Hay muchas formas equivalentes de expresar esta distinción

Agenda

- Historia
- ¿Qué es?

- Predictibilidad
- Inmutabilidad
- Recursion
- Funciones de orden superior
- Modelo de ejecución

- Recursos

Un poco de historia...

- 1958 - Iteración sobre el cálculo lambda lleva a una notación que puede implementarse sobre una computadora existente.
  - Primer lenguaje funcional: LISP - LISt Processing

- Uso original: Programación de modelos matemáticos y de cómputo
  - Cálculo simbólico
  - Lenguajes de programación
  - Algoritmos de optimización
  - Algoritmos de inferencia

- Uso moderno: Sistemas con cambios de estado complejos
  - Programación Concurrente
  - Sistemas Distribuidos

Soporte de Programación Funcional

Lenguajes funcionales dedicados

- Lisp
- Haskell
- Erlang
- Curry
- Idris

Usualmente tienen algún feature que interactúa negativamente con usar funciones no-matemáticas

Soporte de Programación Funcional

Lenguajes multiparadigma

- C++
- Javascript
- Scala
- Python

Usualmente no distinguen entre subrutinas y funciones matemáticas
Es convencional usar las partes funcionales con funciones matemáticas

Soporte de Programación Funcional

Simulado al resolver problemas cuya solución natural es funcional

- Git
- Docker
- REST
- React

¿Qué es?

Paradigma de programación

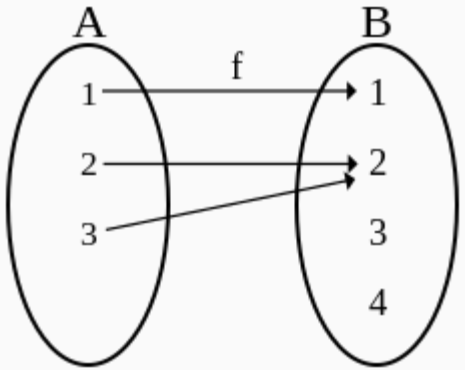
- Basado en funciones matemáticas
- Declarativo

Se piensa en la ejecución de programas como la evaluación de funciones matemáticas.

¿Que es una función?

Función

Para cada elemento del dominio (A), asocia exactamente **UN** elemento del codominio (B)



Puede definirse de muchas formas:

- Tablas
- Expresiones matemáticas
- Subrutina que calcula el valor

Funciones como componentes de primera clase

Las funciones se tratan como cualquier otro valor:

- Tienen expresiones literales

```
function (x, y) { return 2 * x + y }

(x, y) => 2 * x + y
```

- Pueden ser argumentos a funciones
- Pueden ser retornadas por otras funciones

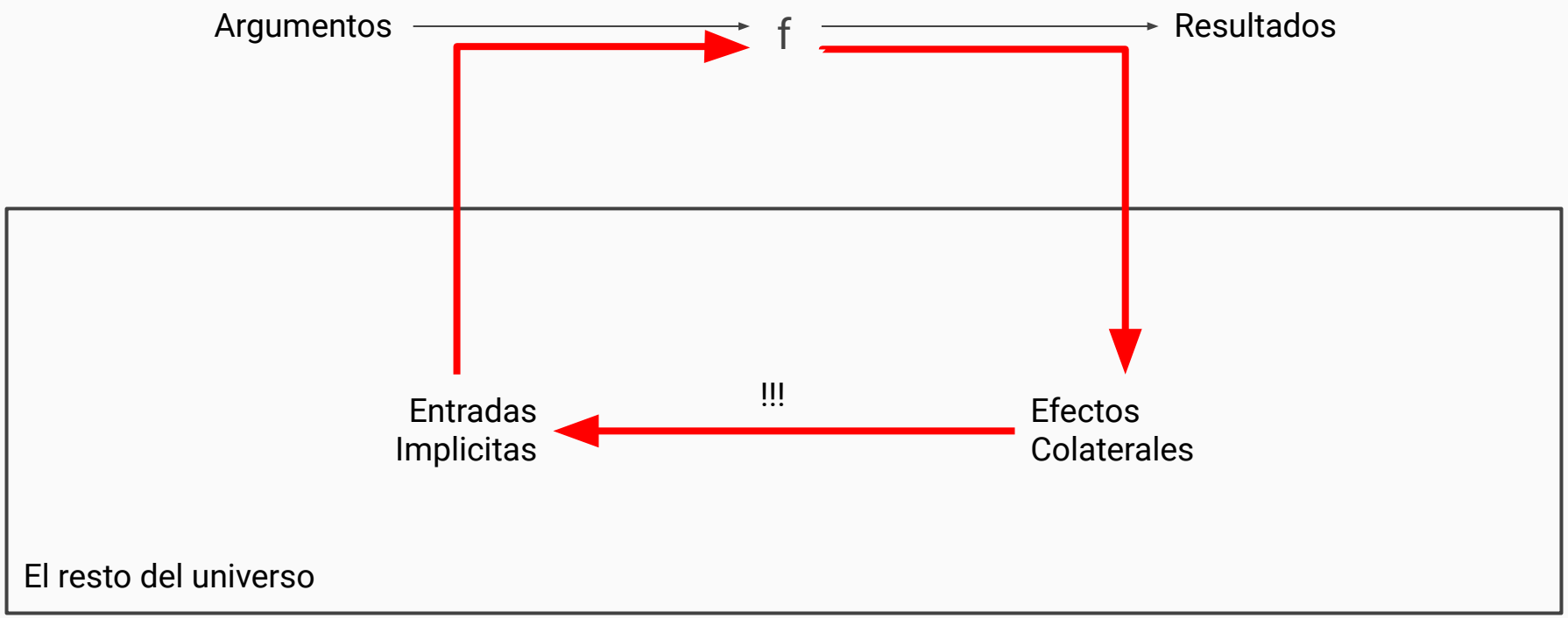
```
const calcularEdadPromedioDe = (condicion) => compose([
    filter(condicion),
    map(persona => persona.edad),
    average
])
```

Programación Declarativa

- **Programación Imperativa**
  - Describe **instrucciones a ejecutarse** paso a paso para variar el estado del programa
  - El estado final debería ser la solución al problema

- **Programación Declarativa**
  - Describe **el problema** que se quiere solucionar en términos de otros problemas
  - El sistema usa esta descripción para intentar hallar una solución

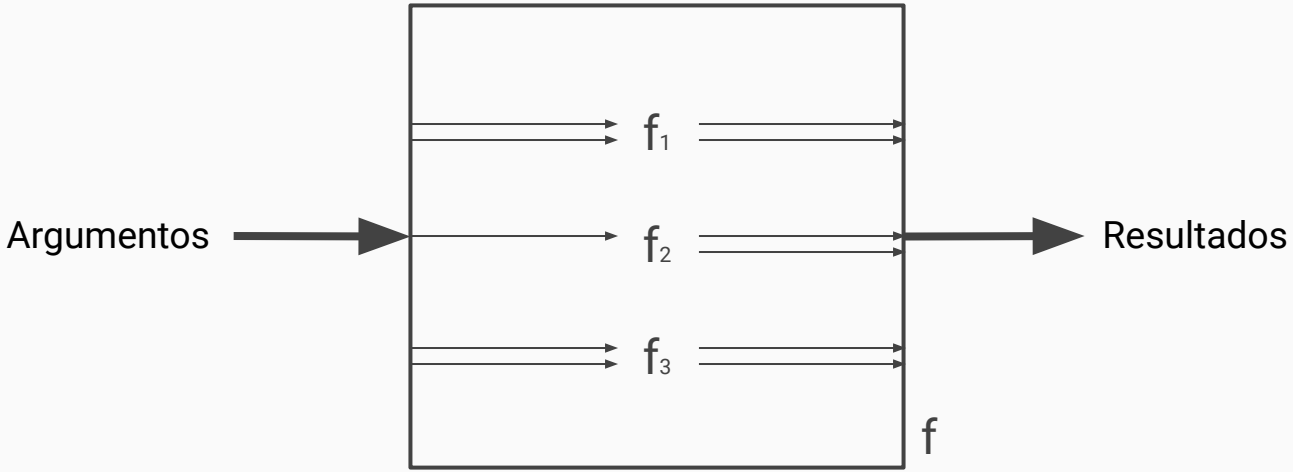
Programación Imperativa



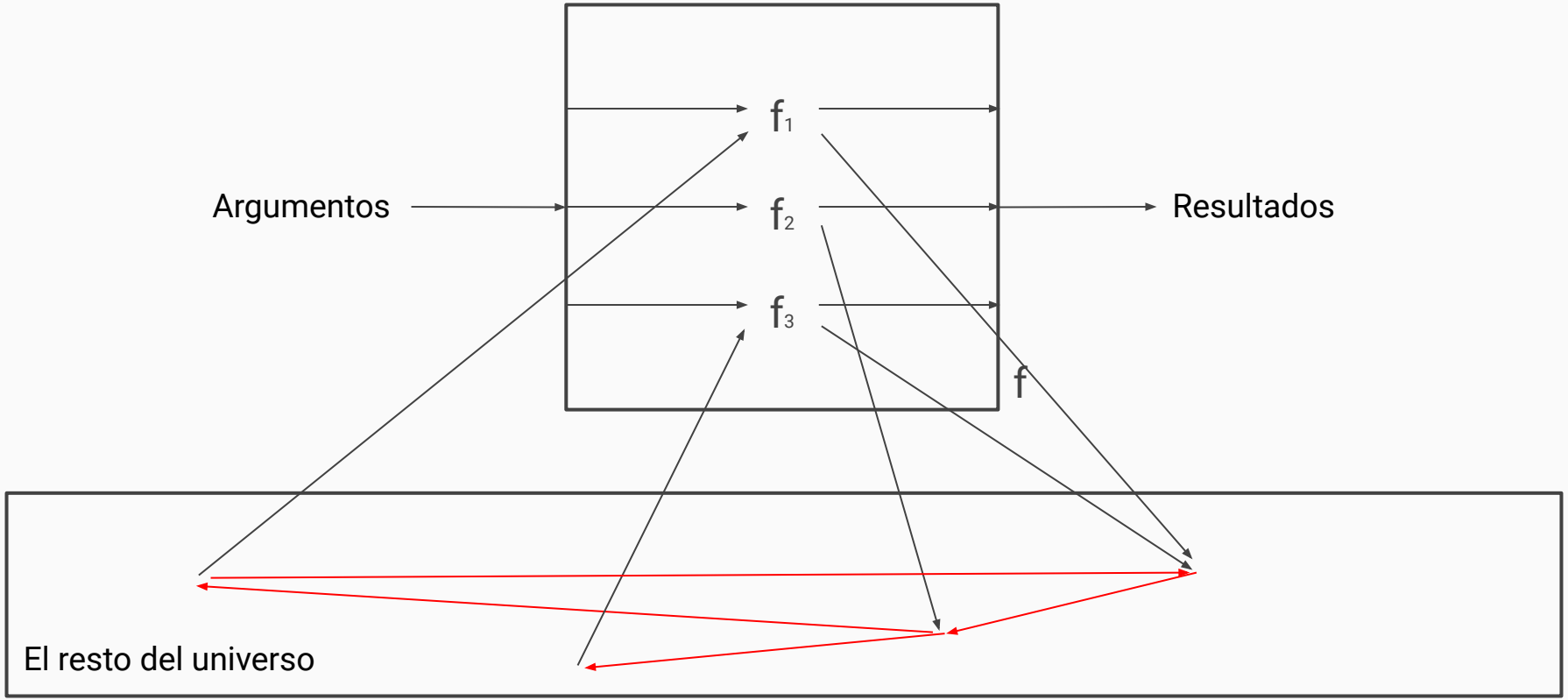
Programación Funcional

Una función matemática está limitada a entradas y salidas en su área local

Composición de funciones (Programación Funcional)



Composición de funciones (Programación Imperativa)



Predictibilidad

Ejecutamos una subrutina repetidas veces:

1ra ejecución: **update() → 0**

2da ejecución: **update() → 1**

¿Qué podemos esperar de una 3ra ejecución?

a) 0 b) 1 c) 2

d) -2 e) 4 f) ?

Predictibilidad

Predictibilidad: Siempre produce la misma salida para una determinada entrada.

Algunas dependencias que rompen predictibilidad:

- Aleatoriedad
- Operaciones Entrada/Salida
  - Red
  - Sistema de archivos
  - Usuario
- Estado variable
  - Tiempo
  - Ubicación
  - Variables globales
    - ➢ Incluyendo Singleton

Predictibilidad

Predictibilidad: Siempre produce la misma salida para una determinada entrada.

Programación Funcional = Programación con Transparencia Referencial

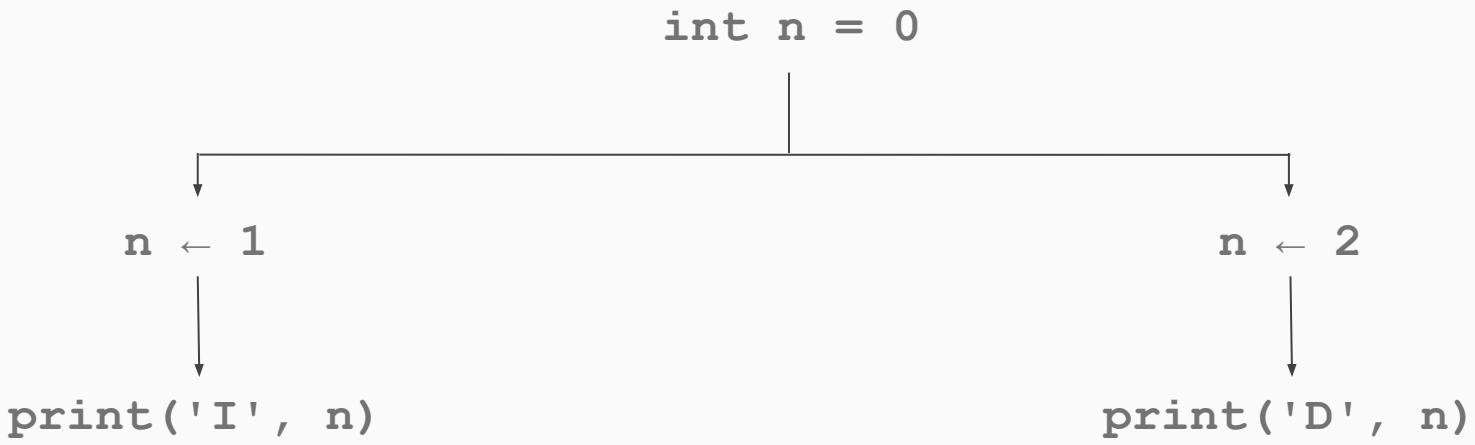
Podemos reemplazar libremente, en cualquier dirección, entre:

- Una expresión (variable o llamado a función)
- Su valor

Inmutabilidad

Dada la siguiente subrutina paralela, ¿qué resultados son posibles?

(Paralelo: Las flechas verticales dan orden de 'sucede después que', no hay otra garantía de orden)



Inmutabilidad

Resultados posibles:

```
I I D D   →  I1 D2

I D I D   →  I2 D2

I D D I   →  D2 I2

D I I D   →  I1 D1

D I D I   →  D1 I1

D D I I   →  D2 I1
```

Inmutabilidad

Inmutabilidad: Evaluar una función no modifica el estado del programa.
En lugar de actualizar valores, se crean nuevos valores.

- Problemas con el Estado Mutable:
  - Condiciones de carrera
  - Accion a distancia
  - Cada mutación puede romper invariantes

- Las funciones no pueden cambiar su entrada o contexto (entradas indirectas)
- En lenguajes más estrictos, no se pueden realizar asignaciones dentro de la definición de una función (Un nombre = Un valor)

Inmutabilidad

Inmutabilidad: Evaluar una función no modifica el estado del programa.
En lugar de actualizar valores, se crean nuevos valores.

Programación Funcional = Programación sin Estado Mutable

- Inmutabilidad ⇒ Predictibilidad
  No hay qué usar para variar los resultados

- Predictibilidad ⇒ Inmutabilidad
  No hay utilidad en implementar mutabilidad si no puede usarse

Recursion

Una entidad es **recursiva** si parte de su definición está dada en términos de la entidad en sí.

Para garantizar que finalice, suele separarse en:

- Caso base (Termina inmediatamente)
- Caso recursivo (Reduce la distancia hacia un caso base)

Recursion

Por inmutabilidad, no podemos usar ciclos como:

```
function sumNumbers (nums: number[]) {
    result = 0;
    for (i = 0; i < nums.length; ++i) {
        result = result + nums[i];
    }
    return result;
}
```

Recursion

En su lugar:

```
function sumNumbers (nums: number[]) {
    if (nums.length == 0) {
        return 0;
    } else {
        return nums[0] + nums.slice(1);
    }
}
```

Recursion

Tail calls: En casos que se llama a otra función e inmediatamente se retorna el valor recibido, ¿es necesario mantener esta función en el stack?

TCO - Tail Call Optimization reemplaza el stack frame en lugar de crear uno nuevo

- ➢ Escribir funciones recursivas sin TCO requiere cuidado con el tamaño del stack

Recursion

```
function sum(x: Nat, y: Nat): Nat {
    if (x == 0) {
        return y;
    } else {
        return sum(x - 1, y + 1);
    }
}

sum(10, 3)
```

Recursion

Stack sin TCO	Stack con TCO
10,3	10,3
10,3 / 9,4	9,4
10,3 / 9,4 / 8,5	8,5
...	...
10,3 / 9,4 / 8,5 / ... / 0,13	0,13

- Si tenemos TCO, usar recursión tiene la misma representación en memoria que un ciclo, pero la mutación pasa a ser detalle de implementación

Funciones de orden superior

Son funciones que **operan sobre otras funciones** como argumento. Se usan para:

- Abstraer estructuras de control
- Describir la intención con la que se manipulan valores
- Tener componentes con propiedades conocidas

Funciones de orden superior

derive(f): Crea una nueva función que evalúa la derivada de f en algún punto.

```
const myCosine = derive(sin)
myCosine(0) = 1
myCosine(pi() / 2) = 0
```

compose(f, g): Composición de funciones

```
const degCosine = compose(toRadians, cos)
degCosine(0) = 1
degCosine(90) = 0
```

Funciones de orden superior

map(coll, f): Crea una copia de **coll** que contiene el resultado de aplicar **f** a cada uno de sus elementos.

```
map([1, 2, 3], inc) → [2, 3, 4]
map([1, 2, 3], isEven) → [false, true, false]
map([a, b, …], f) → [f(a), f(b), …]
```

filter(ls, f): Crea una lista que tiene solo los elementos de **ls** para los que **f** retorna verdadero.

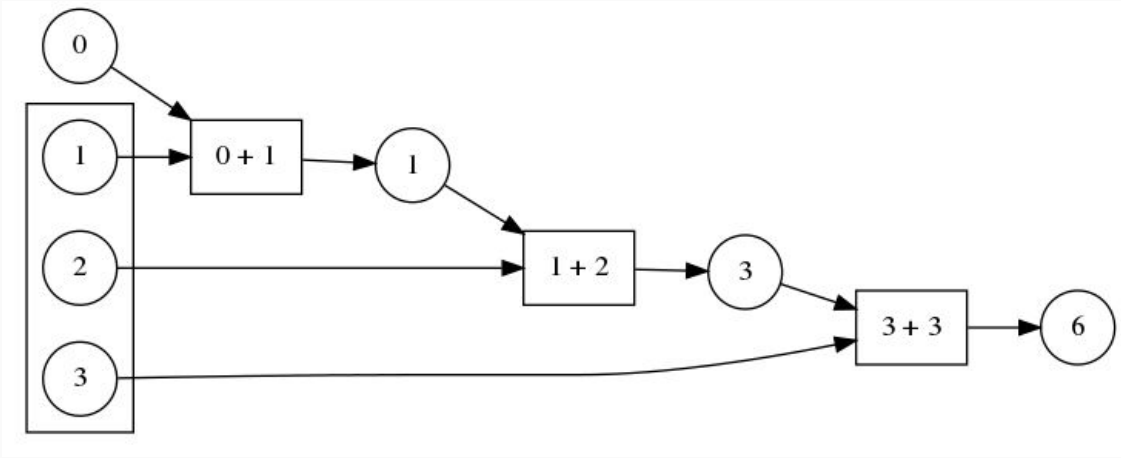
```
filter([1, 2, 3, 4], isEven) → [2, 4]
```

Funciones de orden superior

reduce(ls, f, v0): Usa **f** para combinar un valor inicial **v0** con cada uno de los elementos de **ls**.

```
reduce([1, 2, 3], +, 0) → 6
```

```
0    [1 , 2 , 3]
0 + (1 + 2 + 3)
```



Funciones de orden superior

```
function sumNumbers (nums) {
    return reduce(
        nums,                           // lista
        0,                              // valor inicial
        (acc, curr) => acc + curr       // combinacion
    );
}
```

Se prefiere usar funciones de orden superior específicas en lugar de estructuras de control primitivas:

➢ Funcional usa recursión explicita menos que imperativo usa ciclos

Funciones de orden superior

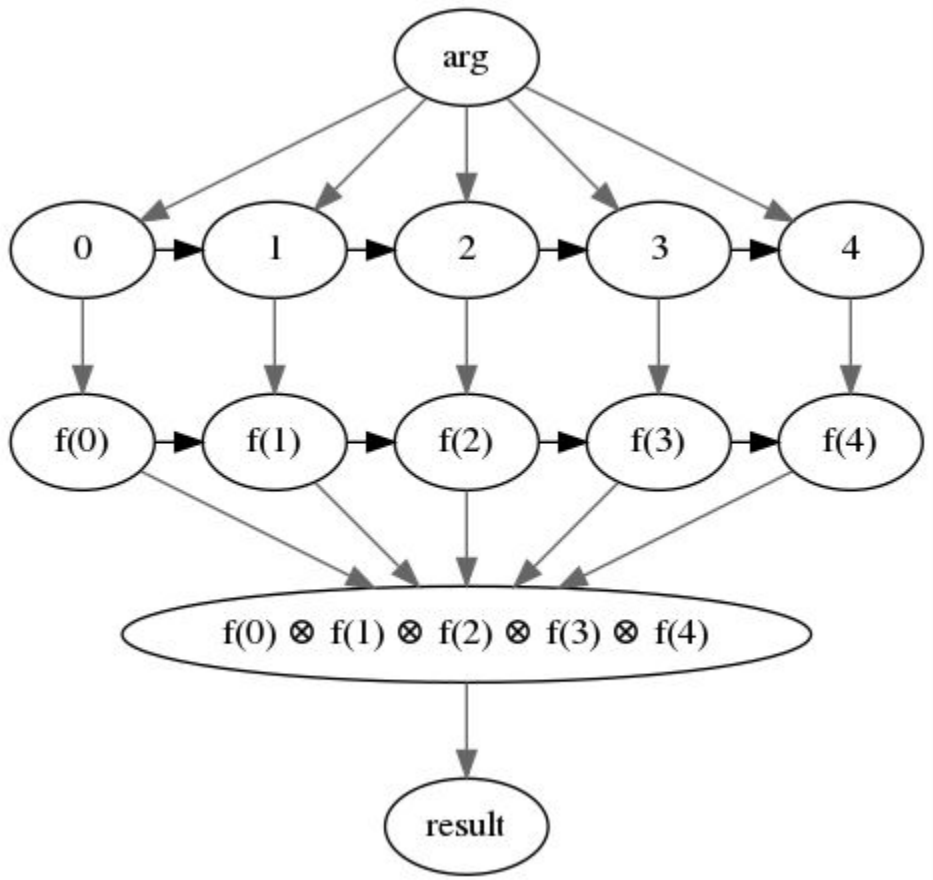
Algunas propiedades (que pueden demostrarse) de map/reduce:

- Por predictibilidad/inmutabilidad
  - Deben dar el mismo resultado independiente de donde se evalúen

- map
  - Debe dar el mismo resultado independientemente del orden en que se evalúan los elementos de la lista

- reduce
  - Si la combinación es asociativa (`x ⊗ (y ⊗ z) = (x ⊗ y) ⊗ z`)
    ⇒ Debe dar el mismo resultado independientemente del orden en que se combinan los pares

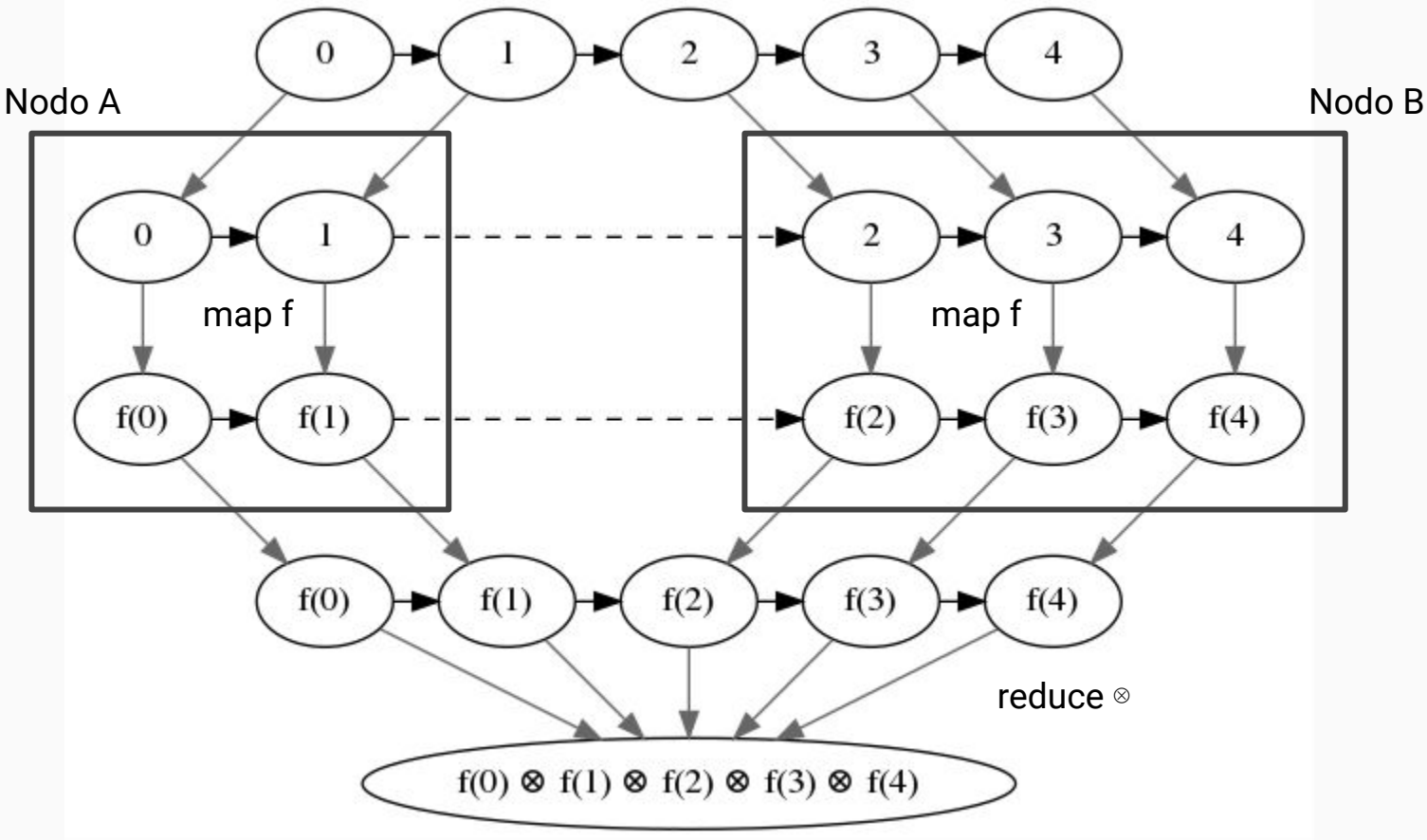
Funciones de orden superior



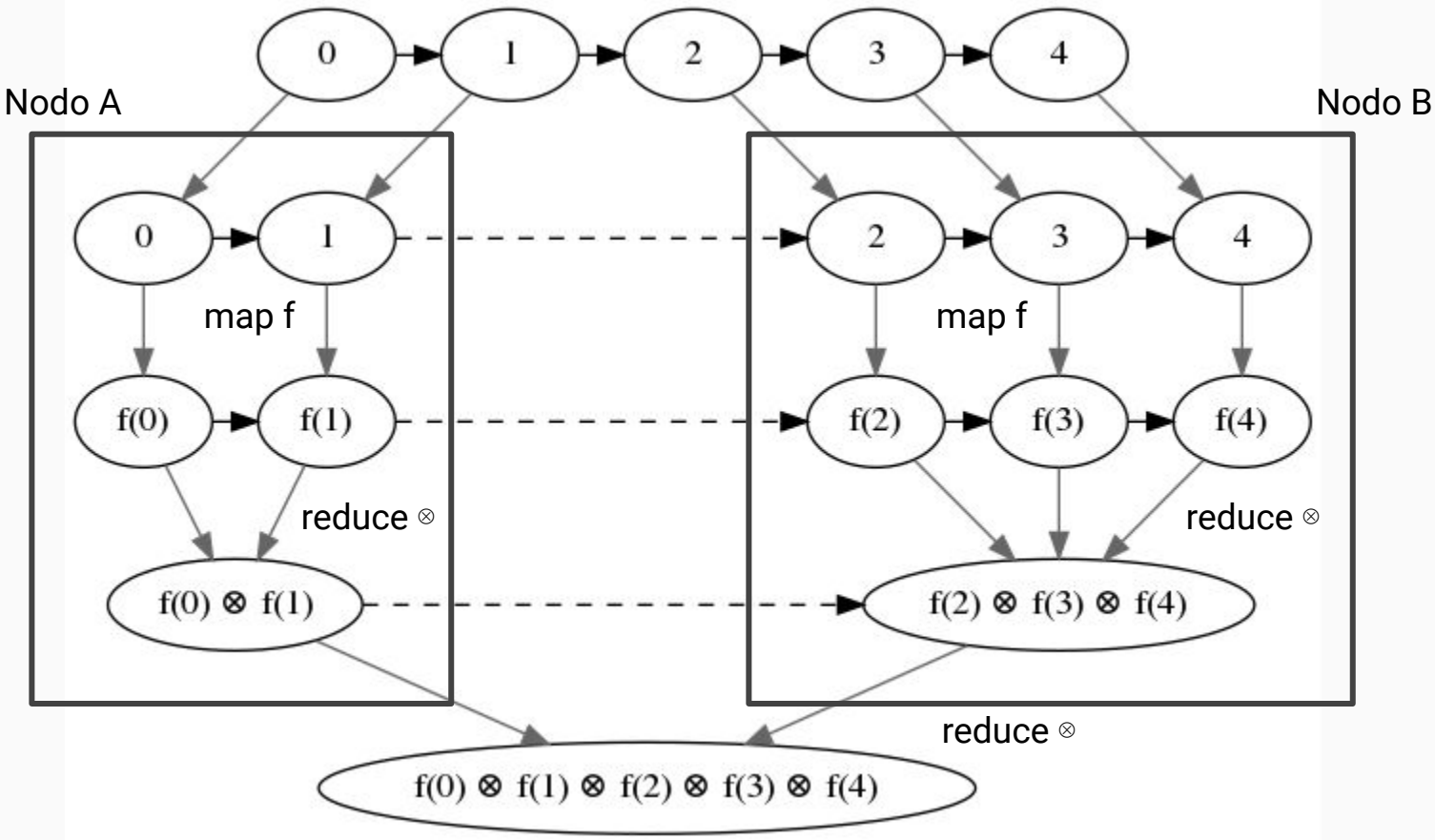
map f

reduce ⊗

Funciones de orden superior



Funciones de orden superior



Preguntas?

Recursos

- Structure and Interpretation of Computer Programs
- Functional Thinking