Programación funcional

Universidad Nacional de Colombia

Lenguajes de Programación

Paradigmas de programación · Sesión teórica

Programación funcional.

Cómo construir programas
con funciones y valores.



Nelson Ivan Castellanos Betancourt

Jay Alvarez Baez

Jesús David Giraldo Gomez

Andres Felipe Moreno Pinzón

David Sebastián Hurtado Sanchez

FUNCIONES Y COMPOSICIÓN

El problema: calcular un reporte de pedidos

Necesitamos sumar los pedidos pagados y reutilizar las reglas cuando cambie el reporte.

PEDIDO	ESTADO	IMPORTE · CENTAVOS COP
A	Pagado	12000
B	Pendiente	9000
C	Pagado	5000
D	Pagado	8000

> **Contrato inicial**
>
> A, C y D aportan 25000 centavos. B queda fuera. Por ahora, los importes ya son enteros válidos.

FUNCIONES Y COMPOSICIÓN

De dónde viene esta forma de programar

Años 1930

Church · cálculo lambda

Un modelo formal de cálculo por abstracción, aplicación y sustitución.

1958 / 1960

Lisp · McCarthy

Implementación desde 1958; artículo en 1960. Procesamiento simbólico para inteligencia artificial.

Años 1970

ML · proyecto LCF

Un metalenguaje para un asistente de pruebas: funciones, tipos y manipulación de expresiones.

1977 / 1978

Backus · composición algebraica

Conferencia de 1977, publicada en 1978: combinar programas y razonar con leyes.

1987 → 1990

Haskell · una base común

Comité en 1987; versión 1.0 en 1990. Unificar la investigación en lenguajes no estrictos.

Estas familias tomaron decisiones distintas sobre tipos, efectos y evaluación; volveremos a ellas al comparar los lenguajes.

FUNCIONES Y COMPOSICIÓN

Una función relaciona entradas y salidas

$$f : A \to B$$

A es el dominio y B es el codominio. Para cada entrada del dominio, una función matemática total asigna una salida.

Importe válido **descuento del 10 %** Importe descontado

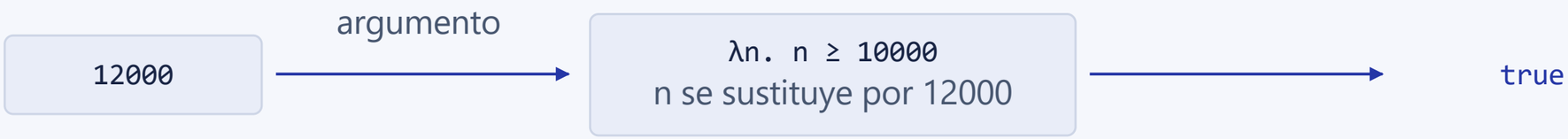
$$f(n) = \left\lfloor \frac{90n}{100} \right\rfloor, \quad n \in \mathbb{N}_0$$

12000 centavos pasan a 10800. El redondeo hacia abajo es parte de esta regla.

Una función de un lenguaje también puede leer archivos, cambiar estado o fallar. En el siguiente bloque precisaremos qué significa que sea pura.

FUNCIONES Y COMPOSICIÓN

Cómo leer una expresión lambda



Esta regla decide si un importe alcanza el mínimo de una promoción.

$$\lambda \underbrace{n}_{\text{parámetro}} . \underbrace{n \geq 10000}_{\text{cuerpo}}$$

Recibe n y devuelve un valor verdadero o falso. El punto separa el parámetro del cuerpo.

$$(\lambda n. n \geq 10000)\ 12000 \longrightarrow 12000 \geq 10000 \longrightarrow \mathbf{true}$$

Aplicamos la función sustituyendo n por 12000. Esa sustitución es una reducción beta.

Usamos números y comparaciones como extensión de la notación lambda. La regla también puede recibir un nombre.

FUNCIONES Y COMPOSICIÓN

La misma regla en tres lenguajes

Scala

```
val elegible =
  (n: BigInt) => n >= 10000
elegible(12000) // true
```

Python

```
elegible = lambda n: n >= 10000
elegible(12000) # True
```

JavaScript

```
const elegible = n => n >= 10000;
elegible(12000); // true
```

Qué estamos comparando

El parámetro es n, el cuerpo es la comparación y el resultado es booleano. Cambia la sintaxis; la regla y esta entrada son las mismas.

FUNCIONES Y COMPOSICIÓN

Podemos pasar esa regla como argumento

Scala

```
val importes = List(BigInt(5000),
  BigInt(12000), BigInt(8000))
val elegible = (n: BigInt) => n >= 10000
importes.filter(elegible)
// List(12000)
```

Python

```
importes = [5000, 12000, 8000]
elegible = lambda n: n >= 10000
list(filter(elegible, importes))
# [12000]
```

Primera clase y orden superior

Una función es de primera clase cuando puede usarse como un valor. filter es de orden superior porque recibe una función: la condición que debe comprobar.

FUNCIONES Y COMPOSICIÓN

La reducción completa, sin saltos

$$g = \lambda n.\, n + 10$$

Nombramos g a la función que suma diez.

$$(\lambda f.\, \lambda x.\, f(f(x)))\ g\ 20\ \rightarrow\ (\lambda x.\, g(g(x)))\ 20$$

1. Sustituimos f por g: queda una función que espera x.

$$(\lambda x.\, g(g(x)))\ 20\ \rightarrow\ g(g(20))\ \rightarrow\ g(30)\ \rightarrow\ 40$$

2. Sustituimos x por 20. Después resolvemos las dos aplicaciones de g.

> **Qué aporta este ejemplo**
>
> La misma función exterior sirve para cualquier transformación compatible. No hemos escrito una regla especial para sumar veinte.

FUNCIONES Y COMPOSICIÓN

Del pedido al total, con los pasos a la vista

Cada pedido tiene id, pagado y centavos. Usamos los cuatro pedidos de la tabla inicial.

Scala

```scala
val pagados = pedidos.filter(p => p.pagado)
// A, C y D
val importes = pagados.map(p => p.centavos)
// List(12000, 5000, 8000)
val total = importes.sum // 25000
```

Python

```python
pagados = [p for p in pedidos if p.pagado]
# A, C y D
importes = [p.centavos for p in pagados]
# [12000, 5000, 8000]
total = sum(importes) # 25000
```

Cómo se obtiene el resultado

Seleccionamos los pagados, extraemos sus importes y los sumamos. map transforma cada elemento; la comprensión de Python expresa aquí la misma transformación.

FUNCIONES Y COMPOSICIÓN

Componer: usar una salida como la siguiente entrada



Scala

```scala
val descuento = (n: BigInt) => n * 90 / 100
val recargo = (n: BigInt) => n + 500
val precioFinal = descuento.andThen(recargo)
precioFinal(12000) // 11300
```

Python

```python
descuento = lambda n: n * 90 // 100
recargo = lambda n: n + 500
precio_final = lambda n: recargo(descuento(n))
precio_final(12000) # 11300
```

$$(g \circ f)(n) = g(f(n))$$

Aplicamos primero el descuento f y después el recargo g: 12000 → 10800 → 11300.

La salida de f debe pertenecer al dominio de g. La división entera redondea hacia abajo para los importes no negativos del ejemplo.

FUNCIONES Y COMPOSICIÓN

Pregunta 1 · ¿qué devuelve el reporte?

Scala

```
pedidos.filter(p => p.pagado).map(p => p.id)
```

JavaScript

```
pedidos.filter(p => p.pagado).map(p => p.id)
```

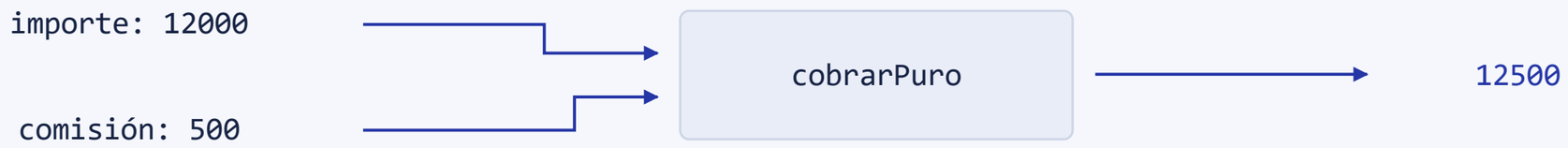
Antes de continuar

A. [A, B, C, D]

B. [A, C, D]

C. 25000

Pureza: ¿de qué depende la salida?



Si una regla lee una comisión global, el mismo importe puede producir cobros distintos.

Scala · comisión externa

```
var comision = BigInt(500)
def cobrar(n: BigInt): BigInt = n + comision
cobrar(12000) // 12500
comision = 900
cobrar(12000) // 12900
```

Scala · comisión como entrada

```
def cobrarPuro(n: BigInt, c: BigInt): BigInt =
  n + c
cobrarPuro(12000, 500) // 12500
cobrarPuro(12000, 900) // 12900
```

Una función pura

Su resultado depende de sus argumentos y no produce efectos observables. La segunda regla recibe la comisión que necesita.

PUREZA, TIPOS Y CATEGORÍAS

Transparencia referencial: sustituir sin cambiar el programa

$$\text{cobrarPuro}(12000, 500) = 12500$$

El cálculo recibe todos los datos que necesita y no modifica el exterior.

Scala · dos expresiones con el mismo comportamiento

```
cobrarPuro(12000, 500) + cobrarPuro(12000, 500)
12500 + 12500  // ambas producen 25000
```

Transparencia referencial

Podemos sustituir la expresión por su valor sin cambiar el comportamiento observable. Un println añadido a cobrarPuro haría que esa sustitución eliminara mensajes.

PUREZA, TIPOS Y CATEGORÍAS

Inmutabilidad: conservar el valor anterior

Scala · List inmutable

```scala
val antes = List(BigInt(12000), BigInt(5000))
val despues = antes :+ BigInt(8000)
// antes: List(12000, 5000)
// despues: List(12000, 5000, 8000)
```

Python · aquí creamos otra lista

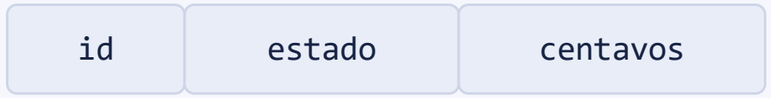
```python
antes = [12000, 5000]
despues = antes + [8000]
# antes: [12000, 5000]
# despues: [12000, 5000, 8000]
```

La operación y el tipo importan

Scala List es inmutable. La lista de Python es mutable, aunque + crea otra lista en este ejemplo. Andres mostrará cómo conservar versiones sin copiar toda la estructura.

Tipos producto y suma: campos y alternativas

Producto: todos los campos



Suma: una alternativa



$\text{Pedido} = \text{Id} \times \text{Estado} \times \text{Centavos}$

Un pedido reúne los tres campos. × significa que necesitamos todos.

$\text{Estado} = \text{Pagado} + \text{Pendiente} + \text{Anulado}$

Un estado elige una alternativa. + distingue casos; no suma dinero.

Scala 3

```
enum Estado:
  case Pagado, Pendiente, Anulado
case class Pedido(id: String,
  estado: Estado, centavos: BigInt)
```

OCaml

```
type estado = Pagado | Pendiente | Anulado
type pedido = { id : string;
  estado : estado; centavos : int }
```

PUREZA, TIPOS Y CATEGORÍAS

Pattern matching: una regla para cada caso

Scala 3

```
def estaPagado(e: Estado): Boolean = e match
  case Estado.Pagado    => true
  case Estado.Pendiente => false
  case Estado.Anulado   => false
val estados = List(Estado.Pagado,
  Estado.Pendiente) // List[Estado]
```

OCaml

```
let esta_pagado = function
  | Pagado -> true
  | Pendiente -> false
  | Anulado -> false
let estados = [Pagado; Pendiente]
(* estado list *)
```

Tipos que se leen y tipos que se deducen

La firma dice Estado → Boolean. El compilador deduce el tipo de estados a partir de sus elementos. Inferir un tipo sigue siendo comprobar tipos estáticamente.

foldLeft: recorrer con un acumulador

Volvemos a los importes ya válidos de A, C y D. Ahora los resumimos en un único total.



Scala · el acumulador y la regla de actualización

```
importes.foldLeft(BigInt(0))((total, importe) => total + importe)
```

Qué abstrae fold

El recorrido se repite; lo que decidimos es el valor inicial y cómo combinarlo con cada elemento. Un fold puede usar otras reglas: no todas permiten reagrupar.

PUREZA, TIPOS Y CATEGORÍAS

El mismo fold en Scala y Python

Scala

```scala
val importes = List(BigInt(12000),
  BigInt(5000), BigInt(8000))
importes.foldLeft(BigInt(0))((a, n) => a + n)
// 25000
```

Python

```python
from functools import reduce
importes = [12000, 5000, 8000]
reduce(lambda a, n: a + n, importes, 0)
# 25000
```

$$a_0 = z, \qquad a_{i+1} = h(a_i, x_{i+1}), \quad 0 \le i < n$$

z es el valor inicial. h recibe el acumulador y el siguiente elemento; devuelve el nuevo acumulador.

El orden es izquierda a derecha. Para reagrupar resúmenes necesitaremos otras propiedades, que explicará Andres.

PUREZA, TIPOS Y CATEGORÍAS

El cálculo puro necesita una frontera con el exterior

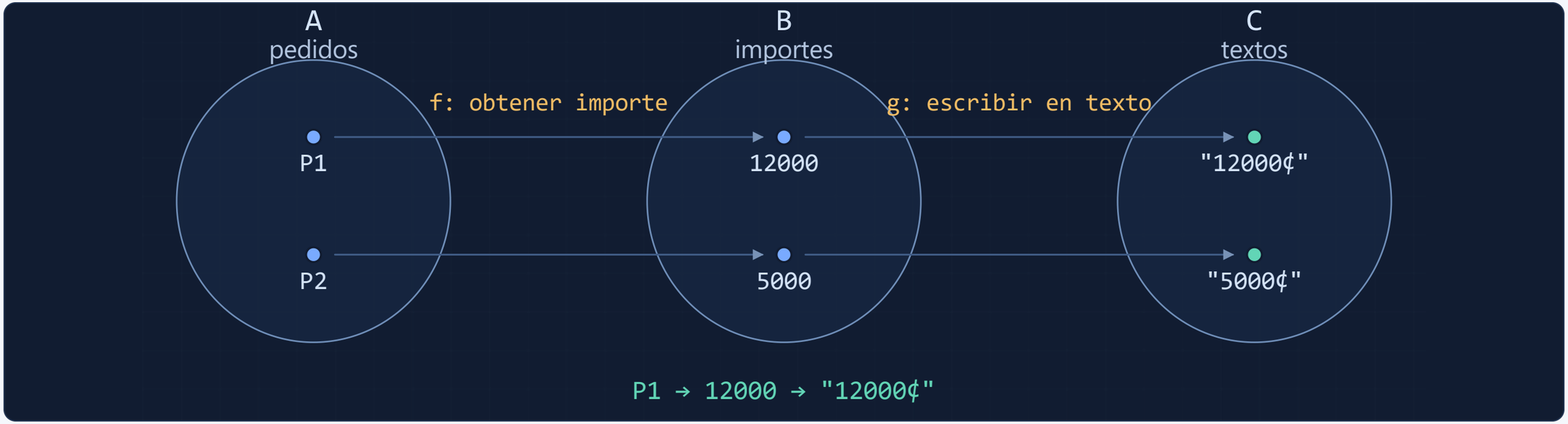


El núcleo y su frontera

Leer archivos y publicar el reporte son efectos observables. En esta primera versión, el núcleo recibe pedidos válidos y devuelve un total. Podemos probar ese cálculo sin tener que leer ni escribir archivos.

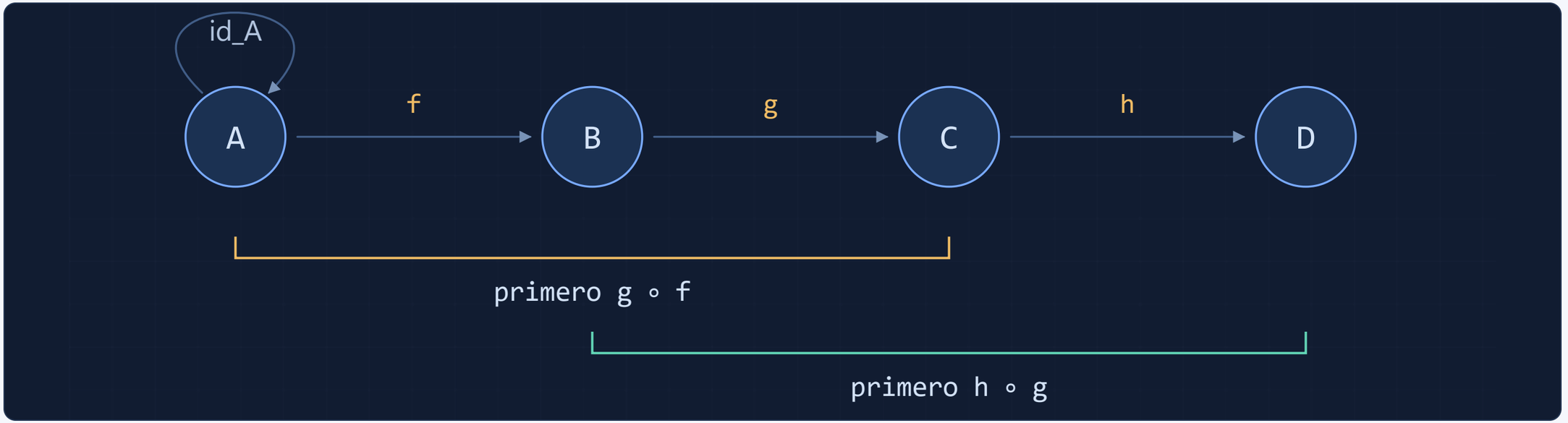
Categorías: empezar con conjuntos y funciones

Un ejemplo es Set: sus objetos son conjuntos y sus morfismos son funciones totales entre ellos.



Un morfismo f: A → B se representa con una flecha. En Set, f es la función completa; las líneas entre puntos muestran sus asignaciones.

Qué exige una categoría



$$h \circ (g \circ f) = (h \circ g) \circ f$$

Asociatividad: la agrupación no cambia el recorrido.

$$\mathrm{id}_B \circ f = f = f \circ \mathrm{id}_A$$

Identidad: componer con la función identidad conserva f.

PUREZA, TIPOS Y CATEGORÍAS

Pregunta 2 · misma entrada, ¿misma salida?

Scala

```
var tasa = 2
val calcular = (n: Int) => n * tasa
val a = calcular(10)
tasa = 3
val b = calcular(10)
```

JavaScript

```
let tasa = 2;
const calcular = n => n * tasa;
const a = calcular(10);
tasa = 3;
const b = calcular(10);
```

Antes de continuar

A. a = 20; b = 20. Es pura.

B. a = 20; b = 30. Es pura.

C. a = 20; b = 30. No es pura.

TRANSFORMACIONES Y ERRORES

Currying: pasar de un par a funciones sucesivas



$$f : A \times B \to C \quad \longmapsto \quad \text{curry}(f) : A \to (B \to C)$$

Scala · dos argumentos → funciones sucesivas

```
val supera: (BigInt, BigInt) => Boolean =
  (limite, importe) => importe > limite
val porLimite = supera.curried

supera(BigInt(10000), BigInt(12000)) // true
porLimite(BigInt(10000))(BigInt(12000)) // true
```

Python · devolver una función

```
supera = lambda limite, importe: importe > limite
por_limite = lambda limite: (
    lambda importe: importe > limite)

supera(10000, 12000)      # True
por_limite(10000)(12000)  # True
```

La condición no cambia: 12000 supera el límite 10000. porLimite(10000) devuelve una función; el segundo paréntesis la aplica a 12000.

Aplicación parcial: preparar una regla para reutilizarla



Scala · funciones como valores

```scala
val supera: (BigInt, BigInt) => Boolean =
  (limite, n) => n > limite
val porLimite = supera.curried
val esGrande = porLimite(BigInt(10000))
List(BigInt(5000), BigInt(12000))
  .filter(esGrande) // List(12000)
```

Python · la misma regla

```python
supera = lambda limite, n: n > limite
por_limite = lambda limite: lambda n: n > limite
es_grande = por_limite(10000)
list(filter(es_grande, [5000, 12000]))
# [12000]
```

$$(\lambda \ell.\, \lambda n.\, n > \ell)\ 10000 \rightarrow \lambda n.\, n > 10000$$

Aplicar solo el límite produce el predicado. Esta aplicación parcial deja pendiente el importe.

La función esGrande conserva acceso a limite = 10000. Esa función con su entorno léxico es una clausura.

TRANSFORMACIONES Y ERRORES

Pregunta 3A · ¿qué función nos queda?

Scala

```
val mayorQue: Int => (Int => Boolean) =
  limite => importe => importe > limite
val esGrande = mayorQue(100)
```

Antes de continuar

¿Qué tipo tiene esGrande después de esta definición?

A. Int → Boolean

B. Boolean

C. (Int, Int) → Boolean

TRANSFORMACIONES Y ERRORES

map y flatMap con las mismas líneas de pedido

P1 contiene teclado y ratón; P2 contiene monitor. Cada pedido produce una lista de productos.

Scala

```scala
val lineas = Map("P1" -> List("teclado", "raton"),
  "P2" -> List("monitor"))
val ids = List("P1", "P2")
val lineasDe = (id: String) => lineas(id)
ids.map(lineasDe)
ids.flatMap(lineasDe)
```

JavaScript

```javascript
const lineas = {P1: ["teclado", "raton"],
  P2: ["monitor"]};
const ids = ["P1", "P2"];
const lineasDe = id => lineas[id];
ids.map(lineasDe);
ids.flatMap(lineasDe);
```

Las dos salidas

map: [["teclado","raton"],["monitor"]]. flatMap: ["teclado","raton","monitor"]. Los identificadores usados existen en lineas.

TRANSFORMACIONES Y ERRORES

Los tipos permiten anticipar el resultado

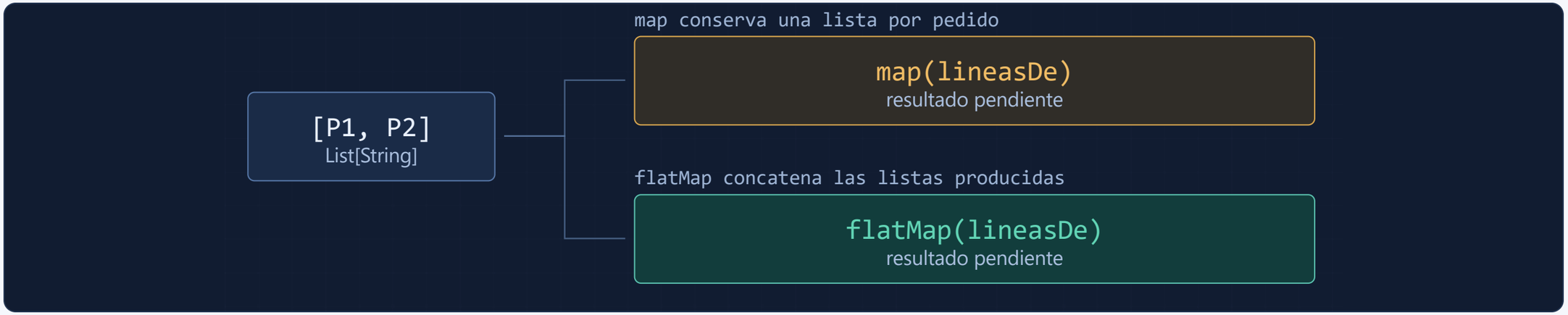
F representa un constructor de tipos. Si F = List, entonces F[A] = List[A].

$$\text{map} : F[A] \to (A \to B) \to F[B]$$

La función recibida devuelve B.

$$\text{flatMap} : F[A] \to (A \to F[B]) \to F[B]$$

La función recibida ya devuelve F[B].

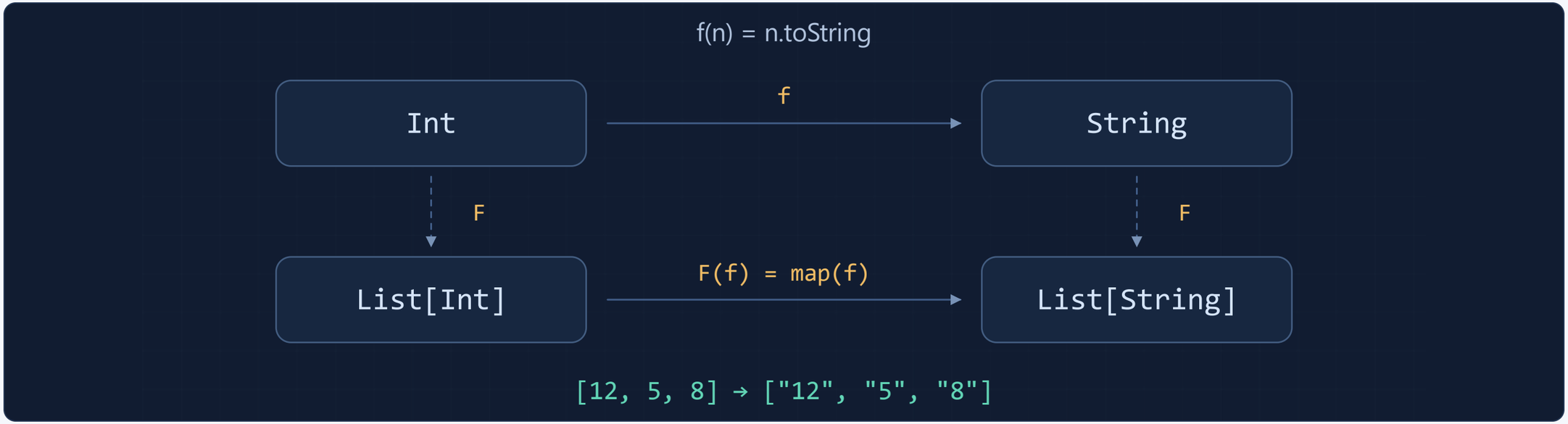


En el ejemplo, lineasDe: String → List[String]. Para map, B = List[String]; para flatMap, F = List y B = String.

TRANSFORMACIONES Y ERRORES

Functor: transformar objetos y morfismos

Un functor F: C → D asigna objetos y morfismos, respetando sus orígenes y destinos. List permite reconocer esa idea en el código.



En este modelo, List actúa dentro de una misma categoría: es un endofunctor. Usamos listas finitas y funciones puras y totales.

TRANSFORMACIONES Y ERRORES

Las dos leyes de Functor

$$F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$$

Transformar con la identidad conserva la lista completa, incluido su orden.

$$F(g \circ f) = F(g) \circ F(f)$$

Transformar con f y después con g equivale a usar su composición.

Scala

```scala
val xs = List(BigInt(12000), BigInt(5000))
val f = (n: BigInt) => n + 500
val g = (n: BigInt) => n.toString
xs.map(f).map(g)
// List("12500", "5500")
```

JavaScript

```javascript
const xs = [12000, 5000];
const f = n => n + 500;
const g = n => String(n);
xs.map(n => g(f(n)));
// ["12500", "5500"]
```

Son igualdades para todas las entradas del contrato. Estos ejemplos permiten comprobar las cuentas; no constituyen una demostración universal.

TRANSFORMACIONES Y ERRORES

Pregunta 3B · ¿este map respeta la identidad?

$$\mathrm{map}(\mathrm{id}, xs) = xs$$

La ley exige conservar xs cuando usamos la función identidad.

Scala

```
def mapRaro[A, B](xs: List[A])(f: A => B): List[B] =
  xs.reverse.map(f)
mapRaro(List(1, 2))(x => x)
```

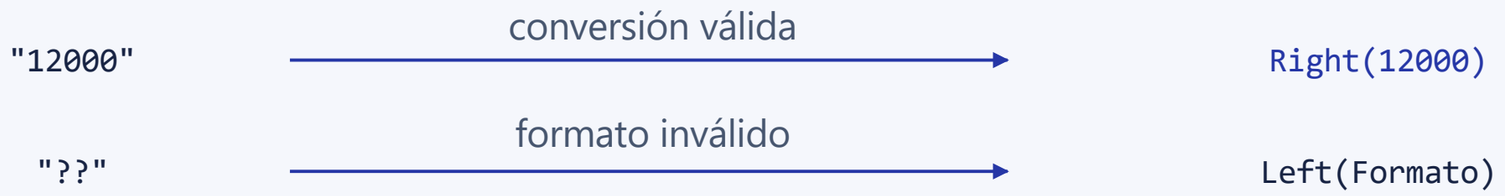
Antes de continuar

A. List(1, 2); sí respeta la ley.

B. List(2, 1); no respeta la ley.

C. List(2, 1); sí respeta la ley.

Un error también puede formar parte del resultado



Ahora recibimos el importe como texto. "12000", "-200" y "??" necesitan respuestas distintas.

$$\mathrm{Either}[E, A] = \mathrm{Left}(E) + \mathrm{Right}(A)$$

Un resultado elige una alternativa: error E o éxito A. Retomamos los tipos suma que vimos antes.

Scala · Either ya existe

```scala
enum Error:
  case Formato, Negativo
val bien: Either[Error, BigInt] = Right(12000)
val mal: Either[Error, BigInt] =
  Left(Error.Formato)
```

OCaml · un tipo equivalente

```ocaml
type error = Formato | Negativo
let bien : (int, error) result = Ok 12000
let mal : (int, error) result = Error Formato
```

Devolver un error puede ser puro

Left o Error son valores. La aplicación decidirá si muestra un mensaje, pide otra entrada o cancela el reporte.

TRANSFORMACIONES Y ERRORES

Validar por casos, en Scala y OCaml

parsear: String → Either[Error, BigInt]

La entrada es texto: parsear("-200") devuelve Right(-200). Después validamos ese número.

Scala · del texto a la validación

```
def noNegativo(n: BigInt):
    Either[Error, BigInt] =
  if n < 0 then Left(Error.Negativo) else Right(n)

parsear("-200") match
  case Left(e) => Left(e)
  case Right(n) => noNegativo(n)
// Left(Error.Negativo)
```

OCaml · los mismos dos pasos

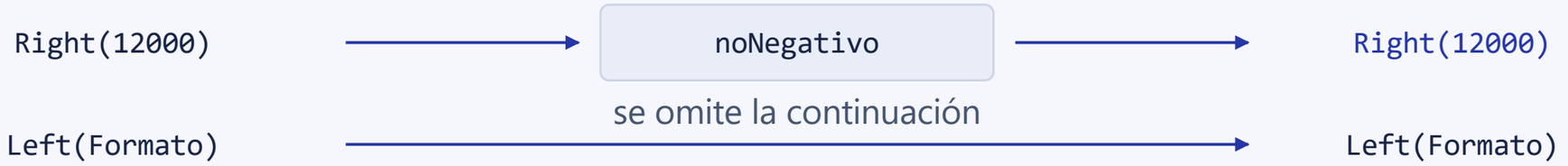
```
let no_negativo n =
  if n < 0 then Error Negativo else Ok n

match parsear "-200" with
| Error e -> Error e
| Ok n -> no_negativo n
(* Error Negativo *)
```

Con "??", parsear devuelve un error de formato y no se llama a noNegativo. La siguiente diapositiva expresa estos mismos casos con flatMap.

También en Rust

Rust incorpora ideas de lenguajes funcionales como OCaml. Su Result<T, E> representa éxito con Ok(valor) y fallo con Err(error): el posible error queda explícito en el tipo del resultado.

Encadenar validaciones con flatMap



Scala

```scala
parsear("12000").flatMap(noNegativo)
// Right(12000)
parsear("-200").flatMap(noNegativo)
// Left(Error.Negativo)
parsear("??").flatMap(noNegativo)
// Left(Error.Formato)
```

OCaml

```ocaml
Result.bind (parsear "12000") no_negativo
(* Ok 12000 *)
Result.bind (parsear "-200") no_negativo
(* Error Negativo *)
Result.bind (parsear "??") no_negativo
(* Error Formato *)
```

El siguiente paso depende del resultado

Con éxito, flatMap llama a noNegativo. Con error, conserva el error y omite esa llamada. map(noNegativo) produciría un resultado anidado.

$$\text{flatMap} : F[A] \to (A \to F[B]) \to F[B]$$

La continuación recibe el valor exitoso y devuelve otro resultado en el mismo contexto.

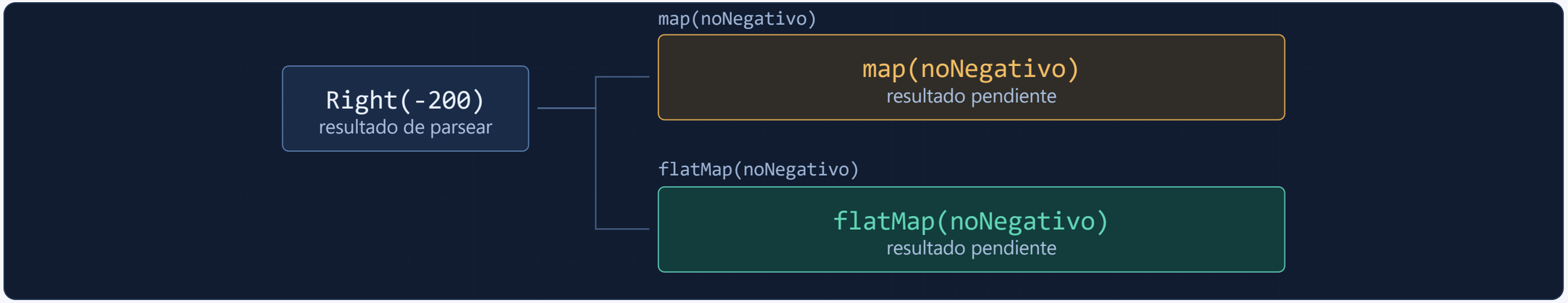
TRANSFORMACIONES Y ERRORES

map y flatMap cuando una validación puede fallar

`parsear: String → Either[Error, BigInt]`

`noNegativo: BigInt → Either[Error, BigInt]`

noNegativo devuelve Left(Negativo) si n < 0; en otro caso devuelve Right(n).



> **La diferencia de tipos**
>
> Con map obtenemos Either[Error, Either[Error, BigInt]]. Con flatMap obtenemos Either[Error, BigInt].

Monad: encadenar con reglas claras

Retomemos f, g y h: cada paso puede fallar. F[X] = Either[E, X], con el mismo tipo de error E.

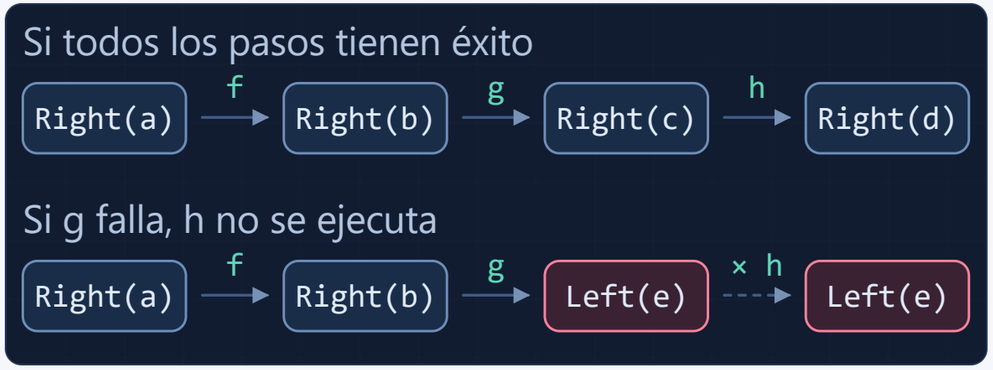
Dos operaciones

$$\text{pure} : A \to F[A]$$

Introduce un valor en el contexto. Aquí, pure(a) = Right(a).

$$\text{flatMap} : F[A] \to (A \to F[B]) \to F[B]$$

Encadena una función que también devuelve un resultado en F.



Tres leyes

Escribimos m >>= f como abreviatura de m.flatMap(f).

$$\text{pure}(a) >>= f = f(a)$$

Identidad izquierda. Introducir a y continuar con f equivale a ejecutar f(a).

$$m >>= \text{pure} = m$$

Identidad derecha. Continuar con pure conserva el resultado m.

$$(m >>= f) >>= g$$
$$= m >>= (\lambda x.\, f(x) >>= g)$$

Asociatividad. Reagrupar conserva el comportamiento; no cambia el orden de los pasos.

Una mónada reúne estas operaciones y sus leyes. Tener un método llamado flatMap no basta.

ESTRUCTURAS Y EVALUACIÓN

Un lote puede contener otros lotes

Trabajamos ahora con importes ya válidos. Un lote puede estar vacío, contener un importe o unir otros dos lotes.

$$L \cong 1 + \mathbb{Z}_{\geq 0} + (L \times L)$$

1 representa el único caso vacío; + indica alternativas; × reúne los dos hijos de Unir.

FORMA	CONTENIDO	RESULTADO
Vacio	Ningún importe	0
Hoja(n)	Un importe válido	n
Unir(i, d)	Dos lotes	Combinar los resultados de i y d

ESTRUCTURAS Y EVALUACIÓN

La función sigue los casos del lote

Scala 3

```
enum Lote:
  case Vacio
  case Hoja(n: BigInt)
  case Unir(i: Lote, d: Lote)
def total(l: Lote): BigInt = l match
  case Lote.Vacio => BigInt(0)
  case Lote.Hoja(n) => n
  case Lote.Unir(i,d) => total(i) + total(d)
```

OCaml

```
type lote = Vacio | Hoja of int
  | Unir of lote * lote
let rec total = function
  | Vacio -> 0
  | Hoja n -> n
  | Unir (i,d) -> total i + total d
```

Qué hace recursiva a la definición

Unir contiene dos lotes. Para calcular su total, aplicamos la misma función a esos sublotes. Vacio y Hoja son los casos que detienen el descenso.

En un lote finito y sin ciclos, cada llamada desciende a un sublote menor. Si los casos base terminan, el recorrido termina; puede mantener O(h) llamadas pendientes para altura h.

ESTRUCTURAS Y EVALUACIÓN

Cómo vuelve el resultado de cada llamada



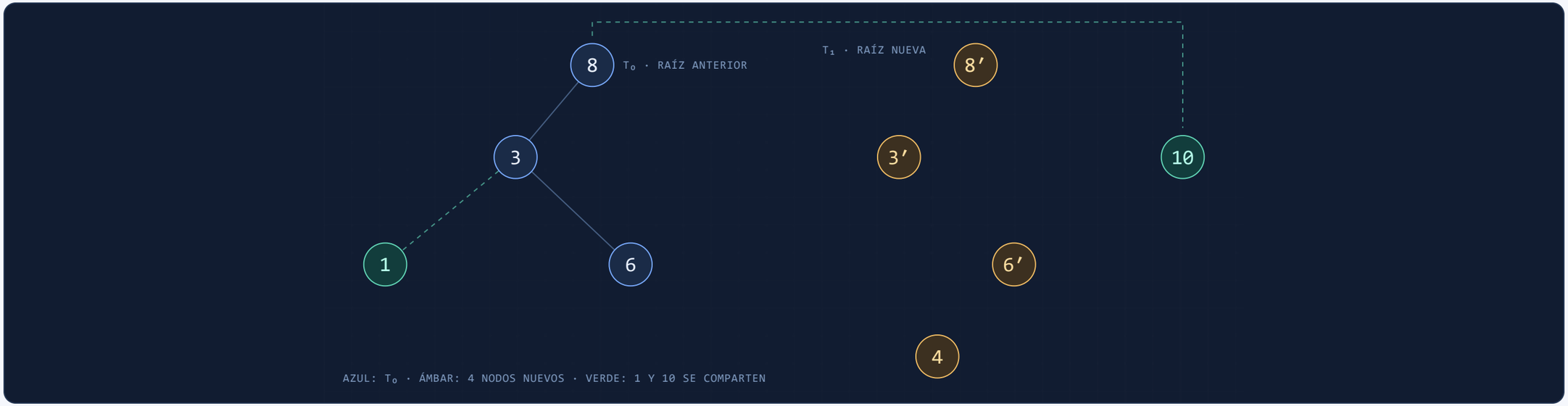
Qué muestra la traza

Una unión espera las respuestas de sus sublotes. Los números de las hojas permanecen visibles y cada total se calcula una sola vez en su estación.

ESTRUCTURAS Y EVALUACIÓN

Persistencia: compartir ramas entre versiones

Una actualización crea otra versión y conserva la anterior. Las ramas intactas se comparten.



Inmutabilidad y persistencia

Insertar 4 crea 4, 6′, 3′ y 8′. Los nodos 1 y 10 se comparten. Las dos raíces siguen disponibles: persistencia aquí significa conservar versiones.

ESTRUCTURAS Y EVALUACIÓN

Qué conserva cada versión del árbol

Scala

```
val anterior = arbol
val nuevo = insert(anterior, 4)
contiene(anterior, 4) // false
contiene(nuevo, 4)    // true
```

Python · implementación inmutable propia

```
anterior = arbol
nuevo = insertar(anterior, 4)
contiene(anterior, 4) # False
contiene(nuevo, 4)    # True
```

El contrato de insertar

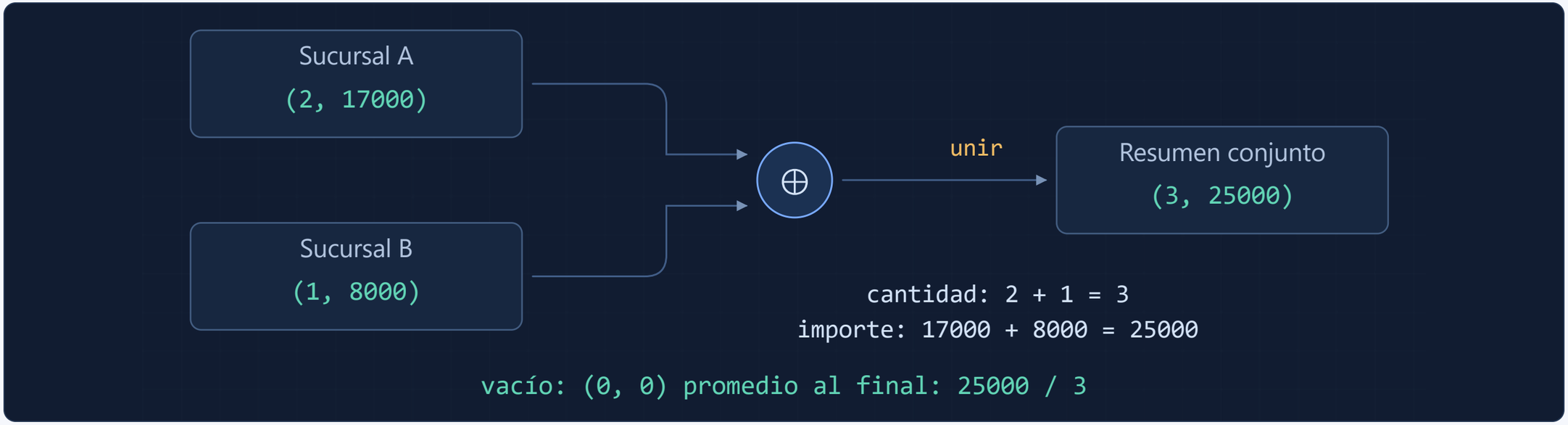
Devuelve otra raíz y comparte las ramas intactas. Las implementaciones completas se incluyen en los ejemplos; no es una propiedad de cualquier árbol de Python.

Insertar copia el camino de búsqueda: O(h) nodos nuevos para altura h. Solo un árbol balanceado garantiza h = O(log n); compartir ramas no garantiza ese balance.

ESTRUCTURAS Y EVALUACIÓN

Combinar resúmenes de distintas sucursales

Queremos conocer cantidad, total y promedio sin volver a recorrer todos los pedidos.



ESTRUCTURAS Y EVALUACIÓN

Monoide: combinar con una identidad y una ley

$$(M, \oplus, e), \qquad \oplus : M \times M \to M, \quad e \in M$$

La operación combina dos elementos de M y devuelve otro de M.

> $$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$
>
> Asociatividad, para todos a, b y c de M.
>
> $$e \oplus a = a = a \oplus e$$
>
> Identidad por ambos lados, para todo a de M.

$$M = \mathbb{N}_0 \times \mathbb{N}_0, \quad (n, s) \oplus (m, t) = (n + m, s + t), \quad e = (0, 0)$$

Nuestro resumen usa cantidad e importe total. Sumamos ambas componentes.

La concatenación de textos también es un monoide, aunque AB ≠ BA. La conmutatividad es una propiedad adicional.

ESTRUCTURAS Y EVALUACIÓN

Del fold al resumen que podemos reagrupar

Scala

```scala
type Resumen = (BigInt, BigInt)
val vacio: Resumen = (0, 0)
def unir(a: Resumen, b: Resumen): Resumen =
  (a._1 + b._1, a._2 + b._2)
val resumen = importes.foldLeft(vacio) {
  (r, n) => unir(r, (BigInt(1), n))
} // (3, 25000)
```

Python

```python
from functools import reduce
vacio = (0, 0)
def unir(a, b):
    return (a[0] + b[0], a[1] + b[1])
resumen = reduce(
    lambda r, n: unir(r, (1, n)),
    importes, vacio) # (3, 25000)
```

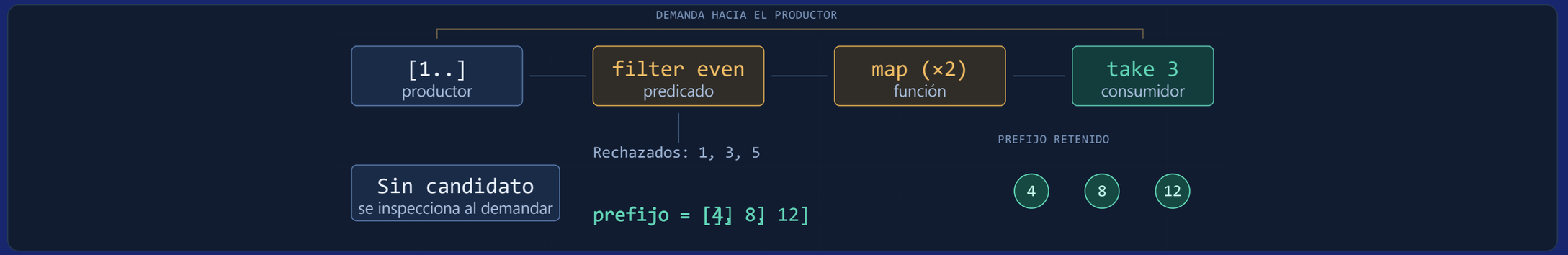
Dos preguntas diferentes

fold determina cómo recorremos. Las leyes del monoide permiten combinar resúmenes de bloques conservando el resultado.

ESTRUCTURAS Y EVALUACIÓN

Evaluación perezosa: calcular lo que se pide

En Haskell podemos pedir un prefijo sin evaluar una secuencia completa.



Haskell

```haskell
take 3 (map (*2) (filter even [1..]))
-- [4,8,12]
```

Scala · secuencia diferida explícita

```scala
LazyList.from(1).filter(_ % 2 == 0)
  .map(_ * 2).take(3).toList
// List(4, 8, 12)
```

Funcional no significa perezoso: OCaml es estricto y Scala evalúa habitualmente por valor. Diferir trabajo también puede retener cálculos pendientes y consumir memoria.

ESTRUCTURAS Y EVALUACIÓN

Pregunta 4 · ¿pedir uno garantiza obtenerlo?

Predecimos la salida usando la condición del filtro.

Haskell

```
take 1 (filter (const False) [1..])
```

Antes de continuar

A. []

B. [1]

C. No produce su primer elemento.

LENGUAJES Y APLICACIONES

Lenguajes con tradición funcional

Las ideas que vimos aparecen en varias familias. Cambian los tipos, la evaluación y la forma de trabajar con efectos.

FAMILIA / LENGUAJES	QUÉ LOS CARACTERIZA
Haskell	Lenguaje funcional puro, tipos estáticos y evaluación no estricta. La entrada/salida se expresa mediante acciones como IO.
Familia ML · OCaml	Funciones, inferencia de tipos, variantes y patrones. OCaml evalúa de forma estricta y también admite objetos y mutación.
Tradición Lisp · Scheme y Clojure	Funciones como valores y alcance léxico. Clojure destaca las colecciones persistentes; ninguno exige pureza en todo el programa.
Erlang y Elixir	Funciones, patrones y datos inmutables. BEAM y su ecosistema aportan procesos, mensajes y herramientas de recuperación.

> **Una distinción útil**
>
> «Funcional» y «multiparadigma» pueden coincidir. Tener una tradición funcional no implica prohibir los efectos ni compartir la misma estrategia de evaluación.

LENGUAJES Y APLICACIONES

Ideas funcionales en lenguajes multiparadigma

Muchos lenguajes incorporan herramientas funcionales y las combinan con objetos, estado y control imperativo.

LENGUAJE	QUÉ INCORPORA DEL ENFOQUE FUNCIONAL
Scala	Funciones de orden superior, colecciones inmutables, tipos algebraicos y patrones; conviven con objetos y estructuras mutables.
Rust	Clausuras, iteradores, enums y patrones. Option y Result hacen explícita la ausencia o el error.
JavaScript	Funciones como valores, clausuras y funciones flecha; map, filter, reduce y flatMap para arreglos.
Python	Funciones como valores, lambda, comprensiones y generadores; itertools y functools para trabajar con secuencias y funciones.
Java / C#	Java combina lambdas con Stream; C# ofrece lambdas y LINQ para expresar consultas y transformaciones declarativas.

> **Qué significa adoptarlas**
>
> Usar una lambda o un map no vuelve pura una función. La pureza depende de lo que hace con sus entradas y con el estado externo. Scala será el puente con la parte práctica.

Ventajas: qué facilita este enfoque

Las ventajas se entienden mejor al relacionarlas con los mecanismos que acabamos de estudiar.

MECANISMO	VENTAJA CONCRETA
Funciones puras	Podemos probar una regla con entradas conocidas y razonar por sustitución, sin preparar una base de datos o un reloj oculto.
Composición y orden superior	Reutilizamos operaciones pequeñas para construir transformaciones más grandes y separar tareas.
Inmutabilidad y persistencia	Conservamos versiones y reducimos cambios inesperados por referencias compartidas.
Tipos algebraicos y patrones	Representamos alternativas y errores de forma explícita; el compilador puede revisar ciertos casos según el lenguaje.
Leyes de las abstracciones	Las leyes de Functor, Monad y monoide permiten justificar equivalencias y reorganizar cálculos bajo sus hipótesis.

> **Alcance**
>
> Estas herramientas ayudan a diseñar y comprobar el programa. Su rendimiento y la corrección de las reglas de negocio todavía requieren medición, pruebas y revisión.

Desventajas y retos al aplicarlo

El enfoque funcional también tiene costos. Conviene reconocerlos antes de elegir una solución.

RETO	QUÉ IMPLICA EN LA PRÁCTICA
Curva de aprendizaje	Leer tipos, currying y abstracciones con leyes exige práctica. Demasiadas capas pueden dificultar la lectura.
Uso de memoria	Las estructuras persistentes comparten partes, pero crear y retener versiones puede aumentar la asignación y la memoria ocupada.
Recursión y evaluación	La recursión puede agotar la pila; la evaluación perezosa puede retener trabajo pendiente. Importan el lenguaje y la implementación.
Integración con efectos	La red, los archivos y la BD siguen presentes. Hay que diseñar sus límites, los reintentos y el tratamiento de fallos.
Bibliotecas, equipo y rendimiento	El ecosistema y la experiencia disponibles condicionan la adopción. Las abstracciones y transformaciones deben medirse en la carga real.

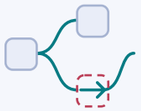
> **Una decisión de ingeniería**
>
> Podemos conservar un núcleo de reglas puras y combinarlo con una capa que gestione la interacción. Un lenguaje multiparadigma permite adoptar el enfoque de manera gradual.

LENGUAJES Y APLICACIONES

Aplicaciones conocidas: de los datos a la interfaz

En estos casos se aprovechan ideas funcionales concretas. Cada sistema también necesita otras decisiones de arquitectura.

DATOS · APACHE SPARK



Procesamiento distribuido

Los RDD son colecciones distribuidas e inmutables. Su historial de transformaciones permite reconstruir particiones perdidas, si los datos son recuperables y el cálculo es determinista.

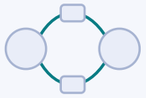
INTERFACES · REACT



Renderizado a partir de datos

El componente calcula la interfaz a partir de sus entradas: props, estado y contexto. Mantener puro el render facilita repetir el cálculo; los eventos y efectos gestionan la interacción.

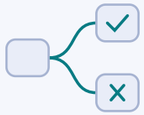
MENSAJERÍA · ERLANG / ELIXIR



Servicios concurrentes

WhatsApp es un uso documentado de Erlang; Discord describe procesos Elixir en su caso de 2023. Datos inmutables y mensajes se combinan con el aislamiento y las herramientas de BEAM/OTP.

FINANZAS · JANE STREET / OCAML



Trading e investigación

Jane Street documenta el uso de OCaml en trading, investigación e infraestructura. Los tipos y la composición ayudan a modelar el dominio; no sustituyen la validación financiera ni las pruebas.

LENGUAJES Y APLICACIONES

Qué nos llevamos a la práctica

Podemos reconocer el enfoque funcional por la forma de organizar el cálculo, incluso dentro de un lenguaje multiparadigma.

01

Describir transformaciones

Funciones como valores, composición y operaciones sobre colecciones permiten construir el cálculo por partes.

02

Hacer explícitos los casos

Los tipos y las operaciones de composición permiten expresar alternativas y continuar según el resultado de cada paso.

03

Controlar el estado y los efectos

La inmutabilidad conserva valores; una frontera clara separa las reglas del dominio de la interacción externa.

> **Conexión con Scala**
> El grupo de práctica mostrará cómo se aplican estas herramientas. Scala combina funciones y objetos: la elección de datos, operaciones y efectos sigue formando parte del diseño.

REFERENCIAS

Referencias

[1] John Hughes (1990 [versión original: 1989]). Why Functional Programming Matters
Introducción y secciones 2–4

[2] Simon Marlow (ed.) (2010). Haskell 2010 Language Report · Introduction
Capítulo 1

[3] Paul Hudak, John Peterson y Joseph Fasel (2000, revisión de junio). A Gentle Introduction to Haskell · Functions
§3, especialmente §3.1 y §3.2

[4] OCaml Documentation (s. f.). Values and Functions
Introduction, Function as Values y Function Application

[5] OCaml Documentation (s. f.). Higher Order Functions
Common Higher-Order Functions; Mapping, Filtering y Folding

[6] Paul Hudak, John Peterson y Joseph Fasel (2000, revisión de junio). A Gentle Introduction to Haskell · Input/Output
§7 y §7.1

Fuentes primarias · consulta: 22–25 de noviembre de 2026.

Referencias · continuación

[7] OCaml Documentation (s. f.). Lists
Functions on Lists y Higher Order Functions on Lists

[8] Cornell University · CS312 (2002). Lecture 21: Side Effects and References
Side Effects

[9] Gang Tan · Penn State (2004). A Brief History of Functional Programming
Their common ancestor: Lambda Calculus

[10] John McCarthy (1979). History of Lisp · The implementation of LISP
The implementation of LISP

[11] Facebook Engineering (2015, 26 de junio). Fighting spam with Haskell
Why Haskell?, Automatic batching and concurrency y How Haskell fits in

[12] Erlang/OTP (s. f.). What is Erlang?
§1.1, §1.3 y §1.5

Fuentes primarias · consulta: 22–25 de noviembre de 2026.

Referencias · continuación

[13] OCaml Documentation (s. f.). Mutability and Imperative Control Flow
Referencias, mutabilidad y control imperativo

[14] Erlang/OTP (s. f.). Expressions
Expression Evaluation y Variables

[15] Simon Marlow (ed.) (2010). Haskell 2010 Language Report · Standard Prelude
PreludeList: map, filter, foldl, foldr y take; composición (.)

[16] Simon Marlow (ed.) (2010). Haskell 2010 Language Report · Expressions
§3.1, §3.3, §3.10 y §3.17

[17] Scala Documentation (s. f.). Tour of Scala · Introduction
What is Scala?, Scala is object-oriented y Scala is functional

[18] Scala Documentation (s. f.). Tour of Scala · Higher-order Functions
Higher-order Functions y Functions that accept functions

Fuentes primarias · consulta: 22–25 de noviembre de 2026.

REFERENCIAS

Referencias · continuación

[19] Scala Documentation (s. f.). Mutable and Immutable Collections
Mutable and Immutable Collections

[20] Scala Documentation (s. f.). Tour of Scala · By-name Parameters
By-name Parameters

[21] Scala Documentation (s. f.). Scala 3 Book · Pure Functions
Pure Functions y But impure functions are needed

[22] Scala Standard Library (s. f.). List · foldLeft
foldLeft[B](z: B)(op: (B, A) => B)

[23] Scala Documentation (s. f.). Algebraic Data Types
Algebraic Datatypes y Recursive Enumerations

[24] Scala Documentation (s. f.). Functional Error Handling
Option, Try y Either

Fuentes primarias · consulta: 22–25 de noviembre de 2026.

Referencias · continuación

[25] Scala Documentation (s. f.). Performance Characteristics
Tablas de secuencias, mapas y explicación de C, eC, Log y L

[26] Chris Okasaki (1998). Purely Functional Data Structures · §2 Persistence
Capítulo 2, pp. 7–16, extracto público del autor/editorial

[27] Haskell base library (s. f.). Data.List · foldl and foldl prime
foldl y foldl': strictness, weak head normal form

[28] Typelevel · Cats (s. f.). Functor
Functor: definición, Identity y Composition

[29] Typelevel · Cats (s. f.). Applicative
pure, ap/product y relación con Functor

[30] Typelevel · Cats (s. f.). Monad
pure, flatMap y flatten

Fuentes primarias · consulta: 22–25 de noviembre de 2026.

Referencias · continuación

[31] Typelevel · Cats (s. f.). MonadLaws · identidades
monadLeftIdentity y monadRightIdentity

[32] Typelevel · Cats (s. f.). FlatMapLaws · asociatividad
flatMapAssociativity

[33] Typelevel · Cats (s. f.). ApplicativeLaws
applicativeIdentity, Homomorphism, Interchange y Composition

[34] John McCarthy (1960). Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I
Introduction; CACM 3(4), 184–195

[35] John Backus (1978). Can Programming Be Liberated from the von Neumann Style?
Conferencia Turing 1977; CACM 21(8), 613–641, introducción y §11

[36] Paul Hudak, John Hughes, Simon Peyton Jones y Philip Wadler (2007). A History of Haskell: Being Lazy With Class
§1–2: comité de 1987 y Haskell 1.0 de 1990

Fuentes primarias · consulta: 22–25 de noviembre de 2026.

Referencias · continuación

[37] OCaml (s. f.). Why OCaml?
A Brief History; Features

[38] John Cowan y comité R7RS (2013; edición HTML 2021). Revised^{7} Report on the Algorithmic Language Scheme
Introduction; §1; §4.1.4; §6.10

[39] LispWorks / ANSI Common Lisp (s. f.). Common Lisp HyperSpec
§3 Evaluation and Compilation; §5 Data and Control Flow; §7 Objects

[40] Microsoft Learn (s. f.). What is F#
Functional programming; data types; interoperability

[41] Clojure (s. f.). Rationale
Functional programming; concurrency; hosted on the JVM

[42] Elixir (s. f.). The Elixir programming language
Maintenance; Scalability; Language features

Fuentes primarias · consulta: 22–25 de noviembre de 2026.

Referencias · continuación

[43] OCaml Documentation (s. f.). Basic Data Types and Pattern Matching
Tuples; Records; Variants; Recursive Variants

[44] Scala Standard Library (s. f.). scala.annotation.tailrec
Descripción de la anotación tailrec

[45] Cornell University · CS3110 (s. f.). Structural Induction
Induction on variants, lists and trees

[46] Typelevel · Cats (s. f.). Monoid
Definición, identidad y ejemplos

[47] Scala Documentation (s. f.). Multiple Parameter Lists
Partial application; currying y funciones sucesivas

[48] Python Software Foundation (s. f.). Python Tutorial · Lambda Expressions
§4.9.6 Lambda Expressions

Fuentes primarias · consulta: 22–25 de noviembre de 2026.

REFERENCIAS

Referencias · continuación

[49] MDN Web Docs (s. f.). JavaScript · Arrow function expressions
Syntax y Function body

[50] Python Software Foundation (s. f.). functools.reduce
functools.reduce(function, iterable, initial)

[51] Emily Riehl (s. f.). Basic Concepts in Category Theory
Definiciones 1.1 y 1.3

[52] MDN Web Docs (s. f.). Closures
Lexical scoping; closures; closure scope chain

[53] Scala Documentation (s. f.). Type Inference
Omitting the type; inference and polymorphism

[54] MDN Web Docs (s. f.). Array.prototype.flatMap
Description y Examples

Fuentes primarias · consulta: 22–25 de noviembre de 2026.

Referencias · continuación

[55] Python Software Foundation (s. f.). List Comprehensions
§5.1.3

[56] Scala Standard Library (s. f.). LazyList
Class documentation; laziness and memoization

[57] OCaml Manual (s. f.). Result
type result; bind; map

[58] Anil Madhavapeddy y Yaron Minsky (s. f.). Real World OCaml · The Compiler Frontend: Parsing and Type Checking
Preprocessing with ppx; The Typed Syntax Tree

[59] Nix & NixOS (s. f.). How Nix Works
Multiple versions; Complete dependencies; Atomic upgrades and rollbacks

[60] NixOS Reproducible Builds (s. f.). NixOS Reproducible Builds
Introducción y Checking reproducibility

Fuentes primarias · consulta: 22–25 de noviembre de 2026.

Referencias · continuación

[61] Ludovic Courtès (2013). Functional Package Management with Guix
Resumen y secciones 2–3

[62] Simon Peyton Jones, Jean-Marc Eber y Julian Seward (2000). Composing contracts: an adventure in financial engineering
ICFP 2000; secciones 3–5 del artículo enlazado

[63] Apache Spark (s. f.). RDD Programming Guide
RDD Operations; Transformations; RDD Persistence

[64] Apache Spark (s. f.). Spark Streaming Programming Guide · Fault-tolerance Semantics
Fault-tolerance Semantics · Background

[65] Discord Engineering (2023, 25 de octubre). Maxjourney: Pushing Discord's Limits with a Million+ Online Users in a Single Server
The challenge; How a message flows; optimizaciones de fanout

[66] Jane Street (s. f.). Technology at Jane Street
Work functionally; Open source

Fuentes primarias · consulta: 22–25 de noviembre de 2026.

Referencias · continuación

[67] Gary Bernhardt (2012, 12 de julio). Functional Core, Imperative Shell
Descripción del diseño y ejemplo del cliente de Twitter

[68] OCaml Documentation (s. f.). Common Error Messages · Pattern Matching Warnings and Errors
This pattern-matching is not exhaustive

[69] Evan Czaplicki y colaboradores (s. f.). Elm compiler · elm.cabal
library y executable; archivos .hs en compiler/src

[70] Lean Language (s. f.). Learn Lean
Introducción y Core documentation

[71] Rocq Prover (s. f.). Rocq Platform
The Rocq Platform

[72] The Rust Project (s. f.). The Rust Reference · Influences
SML, OCaml: algebraic data types, pattern matching, type inference

Fuentes primarias · consulta: 22–25 de noviembre de 2026.

Referencias · continuación

[73] Python Software Foundation (s. f.). Functional Programming HOWTO
Introduction; Iterators; Built-in functions; itertools; functools

[74] Oracle (s. f.). The Java Tutorials · Aggregate Operations
Pipelines and Streams; enlaces a Lambda Expressions y Reduction

[75] Microsoft (s. f.). Language Integrated Query (LINQ)
Introducción y Query expression overview

[76] React (s. f.). Keeping Components Pure
Purity: Components as formulas; Where you can cause side effects

[77] The Rust Project (s. f.). The Rust Programming Language · Functional Language Features: Iterators and Closures
Introducción del capítulo 13

[78] Clojure (s. f.). Functional Programming
First-class functions; Immutable Data Structures

Fuentes primarias · consulta: 22–25 de noviembre de 2026.