.UBAfiuba
FACULTAD DE INGENIERÍA

Programación Funcional

TB025 - Paradigmas de Programación

Curso Essaya

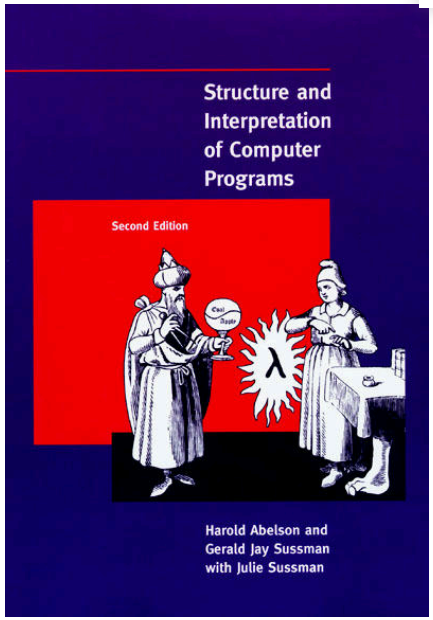
Paradigmas de Programación

- **Imperativos** (énfasis en la ejecución de instrucciones)
  - Programación Procedimental (ej. Pascal)
  - Programación Orientada a Objetos (ej. Smalltalk)
- **Declarativos** (énfasis en la evaluación de expresiones)
  - **Programación Funcional (ej. Haskell)**
  - Programación Lógica (ej. Prolog)

En la Programación Funcional los programas se construyen mediante la aplicación y composición de **funciones**.

Es un paradigma **declarativo** en el que el cuerpo de las funciones contiene **expresiones** en lugar de instrucciones.

Historia



1930s: Cálculo Lambda.

1958 - Lisp: El primero de los lenguajes funcionales, y el más influyente; creado por John McCarthy. De alto nivel, con un sistema de tipos dinámico y una sintaxis basada en listas.

1973 - ML: Precursor de muchos lenguajes modernos. Incorpora un sistema de tipos estático e inferencia de tipos.

1975 - Scheme: Un dialecto de Lisp creado por Guy Steele y Gerald Sussman. En 1985 Sussman y Hal Abelson publican **Structure and Interpretation of Computer Programs** (SICP), que populariza el lenguaje y el paradigma.

1986 - Erlang: Un lenguaje diseñado para sistemas distribuidos y tolerantes a fallos.

1990 - Haskell: Un lenguaje funcional puro, con un sistema de tipos avanzado y evaluación perezosa.

2004 - Scala: Combina programación funcional y orientada a objetos, ejecutándose en la JVM.

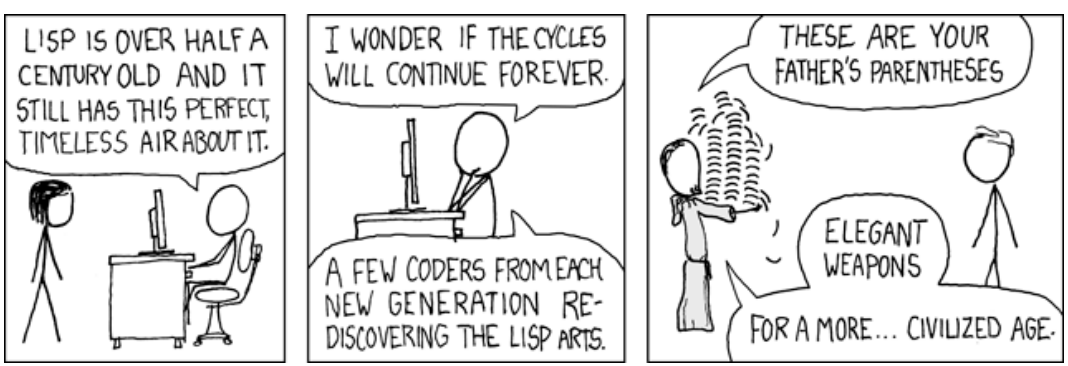
2007 - Clojure: Un dialecto moderno de Lisp que se ejecuta en la JVM.

Los lenguajes modernos como Rust, Java, Kotlin, Swift, etc. incorporan características funcionales, como funciones de orden superior, inmutabilidad y expresiones lambda.

Características

- Funciones puras
- Transparencia referencial
- Datos inmutables
- Las funciones son ciudadanos de primera clase
- Funciones de orden superior
- Recursión
- Evaluación perezosa
- Estructuras de datos funcionales

```
(defun fib (n)
  "Return the nth Fibonacci number."
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))
```



Clojure

```clojure
(defn qsort [l]
  (if (empty? l)
    '()
    (let [f (first l)
          smaller (filter #(<= % f) (rest l))
          bigger (filter #(> % f) (rest l))]
      (concat (qsort smaller) [f] (qsort bigger)))))

(println (qsort [3 6 8 10 1 2 1]))
```

- Creado por Rich Hickey en 2007
- Diseñado para la JVM, interoperable con Java
- Dialecto de Lisp
- Multiparadigma (principalmente funcional)
- Tipado dinámico

Machete de Clojure

Literales:

```
1
nil
true
un-simbolo
:un-keyword
"una cadena"
["un" "vector" "de" "cadenas"]
("una" "lista" "de" "cadenas")
{:tipo "mapa" :clave "valor"}
#{"un conjunto" "de cadenas"}
```

Las listas se interpretan como una **aplicación de función** `(funcion arg1 arg2 ...)`:

```
(+ 1 2) ; => 3
(str "Hola" " mundo!") ; => "Hola mundo!"
(1 2 3) ; => error
'(1 2 3) ; => (1 2 3)
```

Formas especiales:

```
(def pi 3.14159)

(def suma (fn [a b] (+ a b)))

(defn suma [a b] (+ a b)) ; equivalente

(def suma #(+ %1 %2)) ; función anónima

(if (> 3 2)
  (println "Tres es mayor que dos")
  (println "Tres no es mayor que dos"))

(let [x 10 y 20]
  (println "La suma es:" (+ x y)))
```

Funciones puras

Una **función pura** es aquella que cumple con las siguientes propiedades:

- **Comportamiento determinístico:** Para argumentos idénticos siempre produce valores de retorno idénticos.
- **No produce efectos colaterales**.

✘ Impura (no es determinística):

```
(defn tirar-moneda []
  (if (= (rand-int 2) 0)
    :cara
    :ceca))
```

✘ Impura (produce efectos colaterales):

```
(defn saludar [nombre]
  (println "Hola," nombre))
```

✔ Pura :

```
(defn sumar [a b]
  (+ a b))
```

Transparencia referencial

Una expresión es **referencialmente transparente** si puede ser reemplazada por su valor sin cambiar el comportamiento del programa.

Las funciones puras son referencialmente transparentes, lo que permite razonar acerca del código y modificarlo sin efectos inesperados.

Ejemplo: ¿El valor de `y` es el mismo en ambos casos?

```
(def y (+ (f 8) (f 8)))
```

```
(def x (f 8))
(def y (+ x x))
```

Si `f` es una función pura, entonces ambas expresiones son equivalentes (salvando una posible diferencia en eficiencia).

Datos inmutables

> Un programa funcional puro opera exclusivamente con datos inmutables.

En lugar de alterar valores existentes, se crean copias alteradas y se preserva el original.

Ejemplo:

```
(defn agregar-contacto [agenda nombre numero]
  (assoc agenda nombre numero))

(def agenda {"Alan" "1234", "Barbara" "5678"})
(def nueva-agenda (agregar-contacto agenda "Grace" "91011"))

(println agenda)        ; {"Alan" "1234", "Barbara" "5678"}
(println nueva-agenda)  ; {"Alan" "1234", "Barbara" "5678", "Grace" "91011"}
```

Funciones como ciudadanos de primera clase

En la programación funcional, las funciones son **ciudadanos de primera clase**. Esto significa que pueden ser tratadas como cualquier otro valor: pueden ser asignadas a variables, pasadas como argumentos y devueltas como resultados.

```
(let [x 2
      f (fn [y] (+ x y))]
  (f 3))
```

En el ejemplo, la función anónima es una **clausura** (*closure*) que captura el valor de `x` del entorno donde fue definida.

Funciones de orden superior

Una **función de orden superior** es aquella que puede recibir funciones como argumentos o devolver funciones como resultado.

```
(defn dos-veces [f x] (f (f x)))

(println (dos-veces inc 5))  ; 7
(println (dos-veces #(str % "!") "Hola"))  ; "Hola!!"
```

Las funciones `map`, `filter` y `reduce` son ejemplos comunes de funciones de orden superior que operan sobre colecciones.

```
(map inc [1 2 3])         ; (2 3 4)
(filter even? [1 2 3 4]) ; (2 4)
(reduce + [1 2 3 4])      ; 10
```

Recursión

En los lenguajes funcionales, la iteración (ciclos) se logra mediante la **recursión**.

```
(defn _buscar [v x i]
  (if (>= i (count v))
    nil
    (if (= (nth v i) x)
      i
      (_buscar v x (inc i)))))

(defn buscar [v x]
  (_buscar v x 0))
```

Si la llamada recursiva está en **posición de cola**, se puede usar `recur`.

De esta forma, el compilador puede optimizar la llamada recursiva y evitar el crecimiento del *stack*.

```
(defn buscar [v x]
  (loop [i 0]
    (if (>= i (count v))
      nil
      (if (= (nth v i) x)
        i
        (recur (inc i))))))
```

Evaluación perezosa

La **evaluación perezosa** (lazy evaluation) es una estrategia de evaluación de expresiones que retrasa la evaluación de una sub-expresión hasta que su valor sea realmente necesario.

Algunos lenguajes funcionales, como Haskell, tienen evaluación perezosa por defecto.

En otros lenguajes, como Clojure, se puede lograr utilizando estructuras de datos perezosas o funciones específicas.

En Clojure, la mayoría de las funciones que operan sobre **secuencias** son perezosas.

```
(range)           ; (1 2 3 4 5 ...)
(take 5 (range))  ; (1 2 3 4 5)
```

Estructuras de datos funcionales

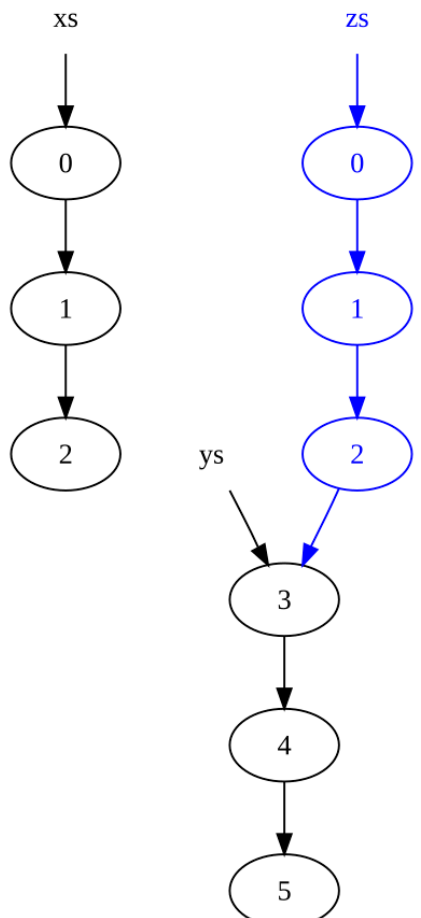
Las estructuras de datos funcionales, o **estructuras de datos persistentes**, son aquellas que siempre preservan la versión anterior ante cualquier modificación.

Estas estructuras de datos son efectivamente **inmutables**.

Las colecciones provistas por Clojure (listas, vectores, conjuntos, mapas, etc.) cumplen con esta propiedad.

Por ejemplo, trabajando con listas enlazadas:

```
(def xs '(0 1 2))
(def ys '(3 4 5))
(def zs (concat xs ys))
```



Ventajas de la Programación Funcional

Código reutilizable: La combinación de funciones puras y de orden superior suele resultar en un código más modular y reutilizable.

Más fácil de razonar: Al evitar el estado compartido y los datos mutables, el comportamiento de una función depende únicamente de sus entradas.

Más fácil de depurar y probar: Las funciones puras son triviales de probar; se proporciona una entrada y se verifica una salida.

Menos código: Los lenguajes funcionales suelen ser más expresivos y concisos, lo que reduce la cantidad de código necesario para lograr el mismo resultado.

Concurrencia más segura: Los datos inmutables son inherentemente seguros para operaciones concurrentes, eliminando categorías enteras de errores comunes como condiciones de carrera y bloqueos.

Familia de lenguajes ML

El lenguaje ML (Meta Language) introdujo en 1973 un sistema de tipos estático e inferido, que permite detectar errores de tipos en tiempo de compilación.

Algunos lenguajes funcionales modernos influidos por ML: Haskell, F#, OCaml, Scala, Rust...

Ejemplo Haskell:

```haskell
data Arbol a = Hoja a | Nodo (Arbol a) (Arbol a)

profundidad :: Arbol a -> Int
profundidad (Hoja _) = 1
profundidad (Nodo izq der) = 1 + max (profundidad izq) (profundidad der)
```

Pattern matching

El lenguaje ML también popularizó el **pattern matching**, una técnica que permite descomponer estructuras de datos complejas de manera concisa y legible.

Ejemplo OCaml:

```
type color = Red | Black
type 'a tree = Empty | Tree of color * 'a tree * 'a * 'a tree

let rebalance t = match t with
    | Tree (Black, Tree (Red, Tree (Red, a, x, b), y, c), z, d)
    | Tree (Black, Tree (Red, a, x, Tree (Red, b, y, c)), z, d)
    | Tree (Black, a, x, Tree (Red, Tree (Red, b, y, c), z, d))
    | Tree (Black, a, x, Tree (Red, b, y, Tree (Red, c, z, d)))
        ->  Tree (Red, Tree (Black, a, x, b), y, Tree (Black, c, z, d))
    | _ -> t
```

Programación funcional en Java

A partir de la versión 8 (lanzada en 2014), Java fue gradualmente introduciendo características funcionales, como expresiones lambda, referencias a métodos y la API de Streams.

Interfaces funcionales: Una **interfaz funcional** es aquella que tiene un único método.

```java
@FunctionalInterface
interface Operacion {
    int aplicar(int a, int b);
}
```

La biblioteca estándar incluye interfaces funcionales comunes como `Predicate`, `Function`, `Consumer`, `Supplier`, etc.

Funciones anónimas: Azúcar sintáctica para crear una clase anónima que implementa una interfaz funcional.

```java
Operacion suma = (a, b) -> a + b;
```

Programación funcional en Java (cont.)

Streams: La API de Streams permite procesar secuencias de datos de manera funcional, utilizando operaciones como `map`, `filter`, `reduce`, etc.

```
List<String> nombres = Arrays.asList("Alicia", "Roberto", "Andrés", "Tomás");
List<String> resultado = nombres.stream()
                            .filter(n -> n.startsWith("A"))
                            .map(String::toUpperCase)
                            .toList();
```

Record classes: Son una forma concisa de definir clases inmutables que actúan principalmente como contenedores de datos.

```
public record Punto(int x, int y) {}

Punto p = new Punto(1, 2);
System.out.println(p.x()); // 1
```

Programación funcional en Java (cont.)

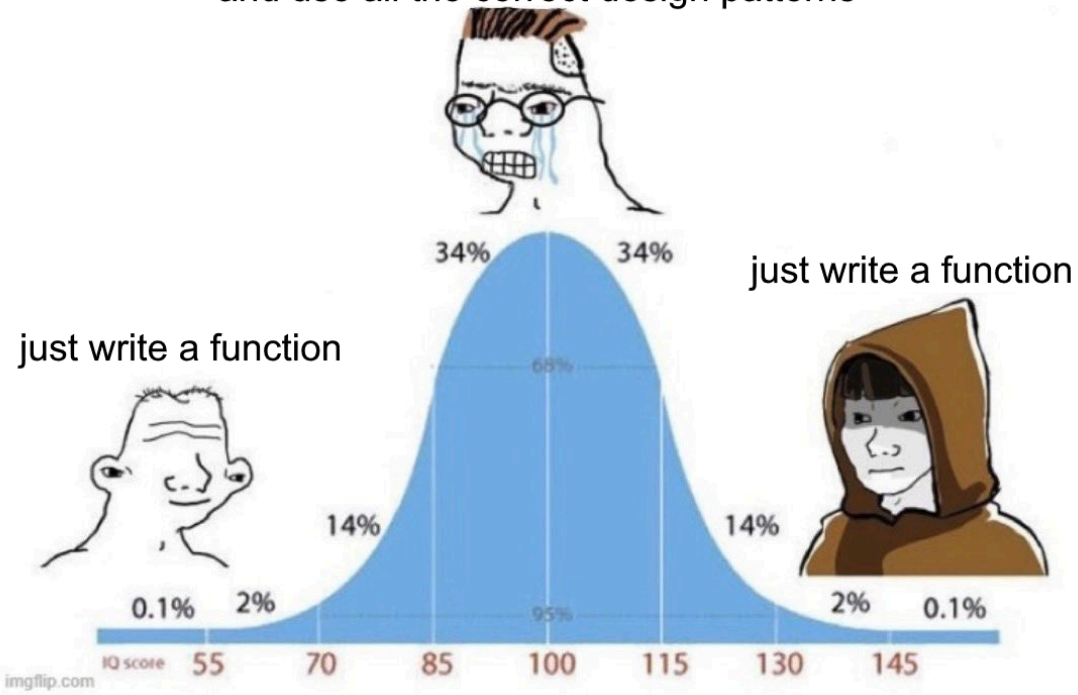
Sealed classes: Permiten definir jerarquías de clases con un control más estricto sobre qué clases pueden extenderse.

```
sealed interface Figura permits Circulo, Cuadrado { ... }

final class Circulo implements Figura { ... }
final class Cuadrado implements Figura { ... }
```

Pattern matching:

```
double calcularArea(Figura f) {
    return switch (f) {
        case Circulo c -> Math.PI * c.radio() * c.radio();
        case Cuadrado q -> q.lado() * q.lado();
    };
}
```


NOOooOOoo you must use an abstract interface to make your code SOLID and DRY and use all the correct design patterns
just write a function
just write a function
34%
34%
14%
14%
0.1%
2%
2%
0.1%
68%
95%
IQ score
55
70
85
100
115
130
145
imgflip.com