# Laboratorio 12

**Problema 1: Teoría**

**1. Reducción-β de NOT (λ-cálculo)**

Usaremos booleanos de Church y mostraremos por β-reducción que NOT TRUE = FALSE y NOT FALSE = TRUE.

- Definiciones:

- TRUE ≡ λa.λb. a
- FALSE ≡ λa.λb. b
- NOT ≡ λp. p FALSE TRUE

Prueba 1: NOT TRUE = FALSE

```
(λp. p FALSE TRUE) TRUE

→ TRUE FALSE TRUE

→ (λa.λb. a) FALSE TRUE      (por definición de TRUE)

→ (λb. FALSE) TRUE           (aplicación con a := FALSE)

→ FALSE                      (aplicación con b := TRUE)
```

Prueba 2: NOT FALSE = TRUE

```
(λp. p FALSE TRUE) FALSE

→ FALSE FALSE TRUE

→ (λa.λb. b) FALSE TRUE      (por definición de FALSE)

→ (λb. b) TRUE               (aplicación con a := FALSE)

→ TRUE                       (aplicación con b := TRUE)
```

Como en ambos casos el resultado coincide con el esperado, la definición de NOT es correcta para los booleanos de Church.

## 2. Recursión y ciclos en programación funcional

### Recursión:

En la programación funcional, los bucles como for o while se sustituyen por funciones que se llaman a sí mismas (recursión).

Ejemplo (JavaScript):

```
   // suma recursiva de una lista

const suma = arr => arr.length === 0 ? 0 : arr[0] +
suma(arr.slice(1));

const numeros = [2, 4, 6, 8];

console.log("Suma total:", suma(numeros)); // 20
```

En este caso, la función suma se llama repetidamente hasta que el arreglo queda vacío.

No hay variables que cambien ni estructuras mutables: cada llamada produce un nuevo resultado.

### Ciclos:

En los lenguajes imperativos, la repetición se hace con for o while, modificando una variable de control.

Ejemplo:

```
let contador = 0;

for (let i = 1; i <= 5; i++) {

  contador += i;

}

console.log("Suma total:", contador); // 15
```

Aquí hay mutación de estado (la variable contador va cambiando).

Este estilo es más simple y eficiente cuando se necesita recorrer muchas iteraciones o controlar el estado

## 3. Cuándo usar la programación funcional (y cuándo no)

La programación funcional es adecuada cuando se busca claridad, inmutabilidad y facilidad para razonar sobre el código. Es útil en tareas de transformación de datos o cálculos puros, donde las funciones no dependen de variables externas ni modifican el estado.

No es recomendable cuando se necesita controlar estados cambiantes, realizar muchas iteraciones intensivas o acceder directamente al hardware, ya que puede volverse menos eficiente.

En resumen, conviene usarla para expresar lógica pura y evitar errores por mutación, pero no en procesos que requieren un control imperativo del flujo o del rendimiento.