# Practica 7 - CPLP

**Ejercicio 1**: Sistemas de tipos:

1. ¿Qué es un sistema de tipos y cuál es su principal función?
2. Definir y contrastar las definiciones de un sistema de tipos fuerte y débil (probablemente
   en la bibliografía se encuentren dos definiciones posibles. Volcar ambas en la
   respuesta). Ejemplificar con al menos 2 lenguajes para cada uno de ellos y justificar.
3. Además de la clasificación anterior, también es posible caracterizar el tipado como
   estático o dinámico. ¿Qué significa esto? Ejemplificar con al menos 2 lenguajes para
   cada uno de ellos y justificar.

1_ Un **sistema de tipos** es un conjunto de reglas que asigna un **tipo de dato** (como entero, cadena, booleano, etc.) a las **variables, expresiones, funciones** y demás elementos de un lenguaje de programación.

### Su principal función es:

**Prevenir errores** en tiempo de compilación o ejecución al garantizar que las operaciones realizadas sobre los datos sean **coherentes con sus tipos**. Por ejemplo, evitar sumar un número con una cadena sin conversión previa.

2_

## Definición 1: (Basada en coerciones implícitas)

### Tipado fuerte:

Un lenguaje tiene tipado fuerte si **no permite operaciones entre tipos incompatibles sin conversión explícita**.

- **Ejemplo**: No se puede sumar un entero y una cadena sin convertir uno de los dos.
- **Coerciones implícitas peligrosas no están permitidas**.

### Tipado débil:

Un lenguaje tiene tipado débil si **permite coerciones implícitas entre tipos incompatibles**, a menudo sin advertencias.

- Esto puede generar errores sutiles o comportamientos inesperados.

---

## Definición 2: (Basada en la robustez del sistema de tipos)

### Tipado fuerte:

Un sistema de tipos es fuerte si **no se puede subvertir fácilmente** el sistema de tipos. Es decir, **una vez que una variable tiene un tipo, este no puede ignorarse o violarse**.

### Tipado débil:

Un sistema de tipos es débil si **permite saltarse o ignorar los tipos de forma sencilla**, por ejemplo mediante conversiones implícitas, punteros, o conversiones inseguras.

---

## Ejemplos de lenguajes

### Lenguajes con tipado fuerte:

| Lenguaje | Justificación |
|---|---|
| **Python** | No permite sumar `int + str` sin conversión explícita. `3 + "4"` da error. |
| **Java** | Las variables tienen tipos fijos. No se puede usar un objeto como otro tipo sin casting explícito y válido. |

### Lenguajes con tipado débil:

| Lenguaje | Justificación |
|---|---|
| **JavaScript** | `"3" + 2` da `"32"`, realiza coerciones implícitas sin advertencias. |

| | |
|---|---|
| **C** | Permite castings peligrosos (por ejemplo, convertir punteros arbitrariamente). |

Resumen: La clasificación de un lenguaje como de tipado fuerte o débil **depende de la definición que se adopte**

C_

## Tipado estático

- La verificación de tipos se realiza **en tiempo de compilación**.
- Los errores de tipo son detectados **antes de ejecutar** el programa.
- Las variables tienen **tipos fijos** y deben declararse explícitamente (o ser inferidos en algunos lenguajes modernos).

## Ejemplos:

| Lenguaje | Justificación |
|---|---|
| **Java** | El tipo de cada variable debe declararse. El compilador detecta errores de tipo antes de ejecutar. |
| **C** | Las variables deben declararse con tipo. El compilador no permite usar variables con tipos incompatibles. |

## Tipado dinámico

- La verificación de tipos se realiza **en tiempo de ejecución**.
- El tipo de una variable puede **cambiar a lo largo del programa**.
- No se requiere declarar el tipo de las variables.

## Ejemplos:

| Lenguaje | Justificación |
|---|---|
| **Python** | El tipo de una variable puede cambiar (ej. `x = 3; x = "hola"`), y los errores de tipo aparecen al ejecutar. |
| **JavaScript** | Las variables no tienen tipo fijo, y pueden recibir valores de distinto tipo sin advertencias del intérprete. |

**Ejercicio 2:** Tipos de datos:

1. Dar una definición de tipo de dato.
2. ¿Qué es un tipo predefinido elemental? Dar ejemplos.
3. ¿Qué es un tipo definido por el usuario? Dar ejemplos.

1_ Un **tipo de dato** es una **categoría de valores** que determina:

- **Qué clase de datos** puede almacenar una variable (por ejemplo, números enteros, texto, booleanos),
- **Qué operaciones** se pueden realizar sobre esos valores (como suma, comparación, concatenación),
- Y **cómo se representa internamente** ese valor en la memoria del computador.

En resumen, el tipo de dato especifica **la naturaleza de los datos y las reglas para manipularlos** dentro de un programa.

2_ Un **tipo predefinido elemental** (también llamado **tipo primitivo**) es un tipo de dato **básico** que está **incorporado en el lenguaje de programación** y que no se define a partir de otros tipos. Sirve como **bloque fundamental** para construir estructuras más complejas.

Estos tipos suelen representar valores simples como números, texto o valores lógicos.

---

## Ejemplos por lenguaje:

## En Java:

- `int`: número entero (ej. `5`)
- `double`: número con decimales (ej. `3.14`)
- `char`: carácter (ej. `'A'`)
- `boolean`: valor lógico (`true` o `false`)

## En Python:

- `int`: número entero

- `float`: número con coma flotante
- `bool`: valor lógico
- `str`: cadena de texto (aunque técnicamente es un objeto, se considera elemental en la práctica)

---

Estos tipos son **gestionados directamente por el compilador o el intérprete**, y forman parte del núcleo del lenguaje.

3_ Un **tipo definido por el usuario** es un tipo de dato **creado por el programador** a partir de otros tipos (elementales o compuestos), con el objetivo de representar **estructuras más complejas o personalizadas** que no están incluidas directamente en el lenguaje.

Permiten **modelar entidades del mundo real** en forma más clara, estructurada y reutilizable.

---

## Ejemplos por lenguaje:

## En Java:

```
public class Persona {
    String nombre;
    int edad;
}
```

Aquí `Persona` es un tipo definido por el usuario que agrupa un `String` y un `int`.

---

## 🔷 En C:

```
typedef struct {
    char nombre[50];
    int edad;
} Persona;
```

Se define un nuevo tipo `Persona` usando `struct`.

---

Este tipo de definición es esencial en la **programación orientada a objetos** y para representar **datos estructurados** en general.

**Ejercicio 3:** Tipos compuestos:

1. Dar una breve definición de: producto cartesiano (en la bibliografía puede aparecer
   también como product type), correspondencia finita, uniones (en la bibliografía puede
   aparecer también como sum type) y tipos recursivos.
2. Identificar a qué clase de tipo de datos pertenecen los siguientes extractos de código.
   En algunos casos puede corresponder más de una:

**Java**

```
class Persona {
  String nombre;
  String apellido;
  int edad;
}
```

**C**

```
typedef struct _nodoLista {
  void *dato;
  struct _nodoLista *siguiente
} nodoLista;

typedef struct _lista {
  int cantidad;
  nodoLista *primero
} Lista;
```

**C**

```
union codigo {
  int numero;
  char id;
};
```

**Ruby**

```
hash = {
  uno: 1,
  dos: 2,
  tres: 3,
  cuatro: 4
}
```

**PHP**

```
function doble($x) {
  return 2 * $x;
}
```

**Python**

```
tuple = ('physics',
'chemistry', 1997, 2000)
```

**Conceptos y Paradigmas de lenguajes de Programación**
**2025**

**Haskell**

```
data ArbolBinarioInt =
  Nil |
  Nodo int
    (ArbolBinarioInt dato)
    (ArbolBinarioInt dato)
```

**Ayuda para interpretar:**
'ArbolBinarioInt' es un tipo de dato que puede ser Nil ("vacío") o un Nodo con un dato número entero (int) junto a un árbol como hijo izquierdo y otro árbol como hijo derecho

**Haskell**

```
data Color =
  Rojo |
  Verde |
  Azul
```

**Ayuda para interpretar:**
'Color' es un tipo de dato que puede ser Rojo, Verde o Azul.

1_

## Producto cartesiano (o product type)

Es un tipo de dato compuesto que **agrupa varios valores, uno de cada tipo especificado**. Se corresponde con la idea de una **tupla** o un **registro**.

**Ejemplo**:

En un `struct` en C o una `class` en Java que combina un `String` y un `int`, el tipo representa el **producto cartesiano** entre los posibles valores del string y los posibles valores enteros.

---

## Correspondencia finita

Es una **asociación entre un conjunto finito de claves y sus valores**. En programación, esto suele representarse como un **diccionario, mapa o arreglo indexado**.

**Ejemplo**:

Un `Map<String, Integer>` en Java, o un `dict` en Python donde las claves son nombres de materias y los valores son sus notas.

---

## Uniones (o sum type)

Es un tipo de dato que puede contener **un valor que pertenece a uno entre varios tipos posibles**, pero **solo uno a la vez**. Es como decir: "esto es un `int` **o** un `String`, pero no ambos".

🔶 **Ejemplo**:

En lenguajes como Haskell o Rust, los `enum` pueden actuar como **sum types**. En C, una `union` también representa esta idea.

```
c
CopiarEditar
union Dato {
    int entero;
    float decimal;
};
```

---

## Tipos recursivos

Son tipos de datos **que se definen en términos de sí mismos**. Se usan para representar estructuras como listas, árboles, etc.

🔶 **Ejemplo**:

```
java
CopiarEditar
class Nodo {
    int valor;
    Nodo siguiente;  // referencia a otro Nodo
}
```

Aquí `Nodo` es un tipo recursivo, porque incluye un campo que es del mismo tipo que la clase.

2_

## Java – `class Persona`

```
class Persona {
    String nombre;
    String apellido;
    int edad;
}
```

**Tipo de producto (product type)**

Agrupa varios campos de distintos tipos (String, int).

**Tipo definido por el usuario**

---

## C – `struct _nodoLista` y `struct _lista`

```
typedef struct _nodoLista {
    void *dato;
    struct _nodoLista *siguiente;
} nodoLista;

typedef struct _lista {
    int cantidad;
    nodoLista *primero;
} Lista;
```

**Tipo recursivo** (por `siguiente` que apunta a otro `nodoLista`)

**Tipo de producto** (combina varios campos)

**Tipo definido por el usuario**

---

## C – `union codigo`

```
union codigo {
   int numero;
   char id;
};
```

**Tipo de unión (sum type)**

Puede ser **int o char**, pero **no ambos a la vez**.

**Tipo definido por el usuario**

---

## Ruby – `hash = { uno: 1, dos: 2, ... }`

```
hash = {
  uno: 1,
  dos: 2,
  tres: 3,
  cuatro: 4
}
```

**Correspondencia finita**

Mapa clave-valor (asociación finita de claves a valores)

---

## PHP – `function doble($x)`

```
function doble($x) {
   return 2 * $x;
}
```

No representa un tipo de dato directamente

Es una **función**, no un tipo compuesto.

---

### Python – `tuple = ('physics', 'chemistry', 1997, 2000)`

```
tuple = ('physics', 'chemistry', 1997, 2000)
```

**Tipo de producto (product type)**

Agrupa varios valores de distintos tipos

---

### Haskell – `data ArbolBinarioInt`

```
data ArbolBinarioInt =
   Nil |
   Nodo int (ArbolBinarioInt) (ArbolBinarioInt)
```

**Tipo recursivo**

**Tipo de unión (sum type)**

Puede ser `Nil` o `Nodo`, lo cual es una alternativa

**Tipo definido por el usuario**

---

### Haskell – `data Color = Rojo | Verde | Azul`

```
data Color = Rojo | Verde | Azul
```

**Tipo de unión (sum type)**

Una variable de tipo `Color` puede ser **uno de varios valores**.

**Tipo definido por el usuario**

**Ejercicio 4:** Mutabilidad/Inmutabilidad:

1. Definir mutabilidad e inmutabilidad respecto a un dato. Dar ejemplos en al menos 2
   lenguajes. TIP: indagar sobre los tipos de datos que ofrece Python y sobre la operación
   #freeze en los objetos de Ruby.
2. Dado el siguiente código:
   a = Dato.new(1)

a = Dato.new(2)
¿Se puede afirmar entonces que el objeto “Dato.new(1)” es mutable? Justificar la
respuesta sea por afirmativa o por la negativa.

1_

## Definición de mutabilidad e inmutabilidad de datos

- **Mutabilidad**: un **dato mutable** puede ser **modificado después de su creación**.
- **Inmutabilidad**: un **dato inmutable no puede modificarse** una vez creado; cualquier cambio produce una **nueva copia** del dato.

---

## Ejemplos en Python

## Mutables:

- `list`, `dict`, `set`

```
lista = [1, 2, 3]
lista[0] = 100  # modifica el primer elemento
```

## Inmutables:

- `int`, `float`, `str`, `tuple`

```
texto = "hola"
# texto[0] = 'H'  # Error: strings son inmutables
texto_nuevo = texto.upper()  # crea una nueva cadena
```

---

## Ejemplos en Ruby

## Mutables:

- Strings por defecto son mutables

```
nombre = "Juan"
nombre[0] = "P"  # ahora es "Puan"
```

### 🔷 Inmutabilidad con `freeze`:

```
nombre = "Juan"
nombre.freeze
# nombre[0] = "P"  # Error: can't modify frozen String
```

`freeze` convierte un objeto en **inmutable**. Si intentás modificarlo, lanza un error.

---

### 📌 Resumen

| Característica | Python | Ruby |
|---|---|---|
| **Dato mutable** | `list`, `dict`, `set` | `String`, `Array`, `Hash` |
| **Dato inmutable** | `int`, `str`, `tuple` | Usando `freeze` en cualquier objeto |

2_

Este código **no modifica** el objeto `Dato.new(1)`; simplemente **reasigna la variable** `a` para que ahora apunte a un **nuevo objeto** (`Dato.new(2)`). El objeto original (`Dato.new(1)`) **sigue existiendo** en memoria (hasta que el recolector de basura lo elimine si ya no hay referencias a él).

---

## Conclusión:

**La mutabilidad/inmutabilidad depende del comportamiento del objeto**, no de la variable que lo referencia.

Este código **no demuestra si el objeto `Dato.new(1)` es mutable o inmutable**, solo que la referencia `a` fue cambiada.

**Ejercicio 5**: Manejo de punteros:

1. ¿Permite C tomar el l-valor de las variables? Ejemplificar.
2. ¿Qué problemas existen en el manejo de punteros? Ejemplificar

1_ Sí, **C permite tomar el l-valor de las variables**, y esto es fundamental para trabajar con **punteros**.
Ejemplo:

```
#include <stdio.h>

int main() {
    int x = 10;         // 'x' es un l-valor porque tiene una dirección en memoria
    int *p = &x;        // se toma el l-valor de 'x' usando '&x'

    printf("Valor de x: %d\n", x);       // Imprime 10
    printf("Dirección de x: %p\n", &x);  // Imprime la dirección de x
    printf("Valor apuntado por p: %d\n", *p);  // Imprime 10, ya que p apunta
a x

    return 0;
}
```

En este ejemplo:

- `x` es un **l-valor** porque puedes obtener su dirección con `&x`.
- `&x` es un **r-valor** (un valor temporal), porque es una dirección, no una ubicación a la que puedas asignar algo directamente.

2_ El manejo de punteros en C es poderoso pero **propenso a errores peligrosos** si no se usa con cuidado. Aquí te detallo algunos **problemas comunes**, con ejemplos:

---

## 1. Punteros no inicializados

Acceder a un puntero que no apunta a una dirección válida.

```
int *p;         // p no está inicializado
*p = 10;        // ERROR: comportamiento indefinido
```

Puede causar un **crash** o sobrescribir memoria aleatoria.

---

## 2. Punteros colgantes (dangling pointers)

Se produce cuando un puntero apunta a memoria liberada.

```
int *p = malloc(sizeof(int));
*p = 5;
free(p);       // Se libera la memoria
*p = 10;       // ERROR: p apunta a memoria ya liberada
```

Puede causar errores difíciles de depurar.

---

## 3. Fugas de memoria (memory leaks)

Ocurre cuando pierdes la referencia a memoria dinámica sin liberarla.

```
int *p = malloc(sizeof(int));
p = NULL;      // La dirección anterior se pierde, no se pudo hacer free
```

La memoria queda ocupada sin poder reutilizarse.

---

## 4. Desbordamiento de punteros (buffer overflow)

Acceso fuera de los límites de un arreglo.

```
int arr[3] = {1, 2, 3};
int *p = arr;
printf("%d\n", *(p + 5)); // ERROR: accede más allá del arreglo
```

Puede corromper datos o permitir vulnerabilidades de seguridad.

---

## 5. Confusión de tipos (type mismatch)

Usar punteros con el tipo incorrecto puede causar resultados inesperados.

```
void *p = malloc(sizeof(int));
*(float *)p = 3.14; // ERROR: la memoria fue reservada para un int, no un flo
at
```

Puede causar lecturas/escrituras erróneas en memoria.

**Ejercicio 6: TAD :**

1. ¿Qué características debe cumplir una unidad para que sea un TAD?
2. Dar algunos ejemplos de TAD en lenguajes tales como ADA, Java, Python, entre otros.

1_ Para que una **unidad** (por ejemplo, una estructura de datos o módulo en programación) sea considerada un **TAD (Tipo Abstracto de Datos)**, debe cumplir con las siguientes **características clave**:

---

## 1. Encapsulamiento

- Oculta los **detalles de implementación interna**.
- Solo se expone al usuario lo que puede hacer (operaciones), no cómo se hace.

*Ejemplo:* Una `Pila` puede usar un arreglo o una lista enlazada internamente, pero el usuario solo ve `push()`, `pop()`, etc.

---

## 2. Definición por comportamiento (no implementación)

- Un TAD se define por el **conjunto de operaciones válidas** y **reglas de uso**, no por el código específico.

*Ejemplo:* Un `Cola` permite `enqueue` y `dequeue` siguiendo el orden FIFO, sin importar si se implementa con arrays o listas.

---

## 3. Independencia de implementación

- La implementación se puede cambiar **sin afectar al usuario**, siempre que se conserven las operaciones y sus efectos.

---

## 4. Operaciones bien definidas

- Cada operación debe estar claramente especificada: qué hace, qué entradas requiere, qué salidas devuelve, y qué efectos secundarios puede tener.

---

## 5. Consistencia interna

- El TAD debe mantener su **estado interno válido** en todo momento tras cada operación.

### 6. Representación interna privada (en lenguajes que lo permiten)

- En lenguajes como Java, C++ o Ada, se recomienda que la estructura interna esté **oculta** usando modificadores como `private`.

---

## Ejemplo breve: TAD Pila (Stack)

**Operaciones:**

- `crear()`
- `push(elemento)`
- `pop()`
- `top()`
- `esVacia()`

**Sin exponer si se usa arreglo, lista, etc.**

2_

# 1. ADA

En Ada, los TAD se definen típicamente como **paquetes** (`package`) que encapsulan el tipo y sus operaciones.

```
package Stack_Pkg is
   type Stack is private;

   procedure Push(S: in out Stack; Element: Integer);
   procedure Pop(S: in out Stack);
   function Top(S: Stack) return Integer;
   function Is_Empty(S: Stack) return Boolean;

private
   type Stack_Array is array (1 .. 100) of Integer;
   type Stack is record
      Data: Stack_Array;
      Top_Index: Integer := 0;
```

```
    end record;
end Stack_Pkg;
```

Aquí, `Stack` es un **TAD**, con operaciones encapsuladas y representación oculta.

---

## 2. Java

En Java, los TAD suelen implementarse como **clases** con atributos privados y métodos públicos.

```
public class Stack {
    private int[] data;
    private int top;

    public Stack(int capacity) {
        data = new int[capacity];
        top = -1;
    }

    public void push(int value) {
        data[++top] = value;
    }

    public int pop() {
        return data[top--];
    }

    public int peek() {
        return data[top];
    }

    public boolean isEmpty() {
        return top == -1;
    }
}
```

Aquí, `Stack` es un TAD porque oculta su implementación y define operaciones bien establecidas.

---

## 3. Python

En Python, los TAD pueden implementarse como clases o incluso usarse directamente desde las **colecciones estándar**.

```
class Stack:
    def __init__(self):
        self._data = []

    def push(self, value):
        self._data.append(value)

    def pop(self):
        return self._data.pop()

    def top(self):
        return self._data[-1]

    def is_empty(self):
        return len(self._data) == 0
```

Aunque Python no fuerza el encapsulamiento, el uso de guiones bajos (`_data`) indica que es un detalle interno.