Práctica 10

1) Un **programa escrito en un lenguaje funcional** se basa en un enfoque completamente diferente al de los lenguajes procedurales. En lugar de una secuencia de instrucciones que modifican el estado de la memoria, **un programa funcional es una composición de funciones puras**, donde cada función recibe unos valores de entrada y devuelve un valor de salida **sin producir efectos secundarios** (como modificar variables globales o estados de memoria).
   En el contexto funcional, **la computadora actúa como un evaluador de expresiones**, al estilo de un sistema matemático. Evalúa funciones aplicadas a argumentos y produce resultados, sin necesidad de modificar estados de memoria ni seguir una secuencia de instrucciones.
   Esto significa que el rol de la computadora es:
   - **Evaluar expresiones funcionales**.
   - **Aplicar funciones y devolver resultados**, sin mantener ni cambiar un estado interno.
   - **Optimizar la ejecución** a través de técnicas como *evaluación perezosa* (lazy evaluation), *memoización*, y *evaluación paralela* (en algunos entornos funcionales).

2) En un **lenguaje funcional**, el lugar donde se definen las funciones se conoce como **entorno** o **contexto léxico**. Este entorno determina **el alcance (scope)** de las variables y funciones.
   Tipos de entornos:
   - **Entorno global:** donde se definen funciones accesibles desde cualquier parte del programa.
   - **Entorno local:** creado cuando se definen funciones dentro de otras funciones (*funciones anidadas*), permitiendo encapsulación.
   - **Entorno extendido:** cuando una función es llamada, se crea un nuevo entorno que extiende el existente, asociando los parámetros formales con los valores reales de la llamada.

3) Una **variable funcional** representa una **asociación simbólica inmutable** entre un nombre y un valor o expresión. Es decir, **una vez que una variable recibe un valor, este no puede cambiar**.
   Al no permitir cambiar el valor de una variable, se evita modificar el estado del programa desde distintos lugares (sin efectos colaterales).
   Una variable puede asociarse no solo a un valor simple, sino también a una expresión (que puede evaluarse más adelante).

4) Una **expresión** en un lenguaje funcional es una **combinación de valores, funciones y operadores** que puede ser evaluada para producir un resultado.
   A diferencia de los lenguajes imperativos, **todo en un lenguaje funcional es una expresión**: incluso las estructuras de control, como `if`, retornan un valor. Una expresión **describe un valor a calcular**, no una acción a ejecutar.

El **valor** de una expresión funcional depende únicamente de los valores de sus **subexpresiones** y de las **funciones involucradas**. Esto se conoce como **transparencia referencial** (Una expresión tiene transparencia referencial si puede ser reemplazada por su valor sin cambiar el comportamiento del programa).

5) **Evaluación por valor**, los argumentos se evalúan antes de llamar a la función. **Evaluación perezosa**, las expresiones se mantienen sin evaluar hasta que realmente se necesiten.

6) Los lenguajes funcionales suelen ser fuertemente tipados. Esto significa que:
   - Cada expresión tiene un tipo bien definido.
   - No se permite mezclar tipos incompatiblemente (por ejemplo, sumar un número con una cadena de texto).
   - Los errores de tipo son detectados en tiempo de compilación, lo que mejora la seguridad y robustez del código.

   Ej:
   - Haskell (estáticamente y fuertemente tipado)
   - OCaml
   - F#

   ¿Por qué?
   - Garantiza seguridad: errores de tipo son detectados antes de ejecutar el programa.
   - Permite inferencia de tipos: sistemas como el de Haskell usan inferencia para deducir automáticamente tipos sin anotarlos.
   - Facilita la composición: al tener tipos bien definidos, las funciones se pueden combinar de forma segura.
   - Favorece la programación declarativa: los tipos ayudan a expresar las restricciones del problema de forma clara.

7) Un programa escrito en **Programación Orientada a Objetos (POO)** está estructurado en torno a **objetos**, que son instancias de **clases**. En lugar de centrarse en funciones o instrucciones (como en los lenguajes imperativos), se basa en la **interacción entre objetos** que representan entidades del mundo real o del dominio del problema. El código se organiza en **clases** que definen cómo deben comportarse los objetos, fomentando la **modularidad, reutilización y escalabilidad** del software.

8) Elementos:
   - Mensajes: Petición de un objeto a otro para que este se comporte de una determinada manera.
   - Métodos: Es un programa que está asociado a un objeto determinado y cuya ejecución solo puede desencadenarse a través de un mensaje recibido por éste o por sus descendientes.
   - Clases: Es un tipo definido por el usuario que determina las estructuras de datos y las operaciones asociadas con ese tipo.
   - Instancia de Clase: Cada vez que se construye un objeto se crea una instancia de esa clase.

- Herencia: Si una clase A tiene una superclase B, entonces A hereda todas las propiedades de B.
- Polimorfismo: Capacidad que tienen los objetos de distintas clases de responder a mensajes con el mismo nombre.

9) **Herencia** (segundo nivel) permite **generalizar comportamientos comunes** en una clase base (superclase) y **especializarlos** en clases derivadas (subclases). Esto introduce un **nivel más alto de abstracción**, ya que permite **modelar jerarquías** y relaciones “es-un”.

10) Tipos:
    - Herencia Simple: Una subclase hereda de **una única clase base**.
    - Herencia Múltiple: Una subclase hereda de **más de una clase base**.
    - Herencia Jerárquica: Varias subclases heredan de **una misma clase base**.
    - Herencia Multinivel: Una clase hereda de una clase que a su vez hereda de otra.
    - Herencia Híbrida: Combina dos o más de los tipos anteriores. Puede generar problemas de ambigüedad si no se maneja correctamente (como el problema del diamante en C++).

    Smalltalk:
    - **Smalltalk solo permite herencia simple.**
    - Es un lenguaje totalmente orientado a objetos, con un modelo limpio y coherente.
    - Cada clase hereda de **una sola clase padre**.
    - La simplicidad fue intencional para mantener la consistencia del sistema de objetos.

    C++:
    - **C++ permite herencia múltiple**, además de todos los otros tipos (simple, jerárquica, multinivel e híbrida).
    - Tiene mecanismos como **herencia virtual** para manejar problemas como el **diamante** (cuando una clase hereda de dos clases que a su vez heredan de una misma clase base).

11) Una **variable** representa un **valor desconocido** que puede ser **unificado** (asignado) con una constante u otra variable durante el proceso de resolución de una consulta.
    Una **constante** representa un **objeto o valor específico del dominio del problema**. Puede ser un número, un átomo (nombre simbólico) o una cadena.

12) Un programa en un **lenguaje lógico**, como **Prolog**, se escribe **declarando hechos, reglas y consultas**. No se indica **cómo** resolver un problema, sino **qué es verdadero** y qué se quiere averiguar.
    Hechos (facts):
    - Describen relaciones que son siempre verdaderas.
    - Se escriben como predicados con argumentos.

    Reglas (rules):

- Describen relaciones **condicionadas**: algo es verdadero **si se cumplen otras condiciones**.
- Se usa `:-` (si).

Consultas (queries):

- Se formulan preguntas al sistema para saber si algo es cierto o para encontrar valores.
- Se escriben en el intérprete con `?-`.

13) **Paradigma Imperativo/Procedural:**

Ej en C:

```
#include <stdio.h>
int main() {
    int numero;
    printf("Ingrese un número entero: ");
    scanf("%d", &numero);

    if (numero % 2 == 0) {
        printf("El valor ingresado es PAR\n");
    } else {
        printf("El valor ingresado es IMPAR\n");
    }

    return 0;
}
```

Paradigma Lógico:

Ej en Prolog:

```
% Hechos y reglas
par_o_impar(N) :-
    N mod 2 =:= 0,
    write('El valor ingresado es PAR'), nl.

par_o_impar(N) :-
    N mod 2 =\= 0,
    write('El valor ingresado es IMPAR'), nl.

% Consulta simulada:
% ?- par_o_impar(7).
% El valor ingresado es IMPAR
```

Paradigma Funcional:

Ej en Haskell:

```
main :: IO ()
main = do
```

```
    putStrLn "Ingrese un número entero:"
    input <- getLine
    let numero = read input :: Int
    putStrLn (parImpar numero)

parImpar :: Int -> String
parImpar n
    | even n    = "El valor ingresado es PAR"
    | otherwise = "El valor ingresado es IMPAR"
```

---

14) Los **lenguajes de scripting** (o lenguajes basados en scripts) son lenguajes de programación diseñados principalmente para **automatizar tareas**, **controlar otros programas** o **realizar tareas repetitivas de manera eficiente**, con un enfoque en la **simplicidad y rapidez de desarrollo**, más que en el rendimiento o control detallado del hardware.
    Características:
    - Son interpretados
    - Por lo general tienen tipado dinámico
    - Sintaxis simple y flexible
    - Alta productividad

    Ejemplos:
    - Python
    - Perl
    - Javascript
    - Bash
    - PHP

15) Existe otra clasificación para los paradigmas:
    - **Dirigido por Eventos:** El flujo del programa está determinado por sucesos externos. La ejecución se desencadena por eventos externos (clics, respuestas de red, etc.).
        - Muy usado en desarrollo web y aplicaciones gráficas.
    - **Orientado a Aspectos:** Apunta a dividir el programa en módulos independientes, cada uno con un comportamiento y responsabilidad bien definido. Permite separar **funcionalidades transversales** (como logging, seguridad) del código principal.
        - Ejemplo: AspectJ (extensión de Java).
    - Existen otros paradigmas no mencionados, como: paradigma basado en componentes, paradigma reactivo, etc.