**Simulacro de Coloquio Final - Avanzado**

## Ejercicio 1: Programación Lógica en Prolog

Dado el siguiente conocimiento en Prolog:

```
% Todos los pájaros vuelan excepto los que están heridos
vuela(X) :- pajaro(X), \+ herido(X).

% Si un animal es mamífero, no es un pájaro
pajaro(X) :- animal(X), \+ mamifero(X).

% Tweety es un pájaro
pajaro(tweety).

% Tweety está herido
herido(tweety).

% Los pingüinos no vuelan
vuela(X) :- pingüino(X), false.

% Pingu es un pingüino
pingüino(pingu).

% Garfield es un mamífero
mamifero(garfield).

% Garfield es un animal
animal(garfield).
```

1. **a)** ¿Quiénes pueden volar según la base de conocimiento? Realice las consultas necesarias y justifique cada respuesta.
   **b)** Agregue una regla que permita registrar los animales que están siendo tratados (curados) y, una vez curados, puedan volar.

   a) De la consulta vuela(x). A su vez tenemos las siguientes reglas: si un animal es mamífero, no vuela (pajaro(X) :- animal(X), \+ mamifero(X).), (Todos los pájaros vuelan excepto los que están heridos) vuela(X) :- pajaro(X), \+ herido(X). Prolog busca en su base un X tal que X satisfaga a la consulta, entonces de pájaro(tweety) y Herido(Tweety), se cumple un x tal que vuela(tweety) :- pájaro(tweety), \+ herido(tweety). Además con una búsqueda similar, encontramos que pájaro(garfield) :- animal(garfield), mamífero(garfield). Sumado a la cláusula de que los pingüinos no vuelan, encontramos en la consulta que ningún animal del programa puede volar de momento.

   b) vuela(x):- pajaro(x), en_tratamiento(x)

   tratado(x):- herido(x), en_tratamiento(x)

## Ejercicio 2: Programación Funcional en Clojure

Considere el siguiente código en Clojure:

```
(ns simulacro.core)

(defn generate-nums [n]
  (lazy-seq (cons n (generate-nums (+ n 1)))))

(def nums (generate-nums 1))

(defn filter-odd-even [nums]
  {:odd (filter odd? nums)
   :even (filter even? nums)})

(defn parallel-computation []
  (let [result (atom {:odd [] :even []})]
    (doseq [n (range 1 10)]
      (future
        (swap! result update-in [:odd] conj (* n 2))
        (swap! result update-in [:even] conj (+ n 3))))
    @result))
```

1. **a)** Explique el propósito de las funciones `generate-nums`, `filter-odd-even`, y `parallel-computation`.
   **b)** ¿Qué ocurre si se cambia `swap!` por `reset!` en la función `parallel-computation`? Justifique el impacto en el resultado.

   a) Tenemos por un lado a generate-nums, que esta generando una secuencia infinita de números, tales que n comience desde el 1. De esto, filter-odd-even esta filtrando esa secuencia entre los pares y los impares de la secuencia y guardandolas en dos sublistas, que respondan esos dos criterios. Por último, gracias al rango del 1 al 10 que se le da a esta secuencia en parallel-computation, se genera en base a esas dos sublistas creadas y ya definidas en [1 2 3 4 5 6 7 8 9 10], dos cálculos para cada criterio. Esto lo hace primero guardando los valores atómicos en una lista vacía para mapear los valores definidos, y así evitar mutabilidad. Y a su vez gracias al swap! se cambian cada valor de estas listas odd y even en base al cálculo establecido para cada elemento n de estas. Así quedaría odd:[2 4 6 8 10 12 14 16 18 20]; Y Even: [4 5 6 7 8 9 10 11 12 13].

   b) Se produciría una potencial condición de carrera ya que el reset intentaría cambiar los valores anteriores al cálculo y sobreescribirlos, produciendo una potencial sobreescritura (sobre todo en el último valor) y problemas con ambos hilos. Además causaría que cada hilo sobreescriba el estado del `atom`, eliminando los resultados previos y generando resultados inconsistentes, especialmente cuando varios hilos escriben al mismo tiempo.

## Ejercicio 3: Principios de Diseño

Dado el siguiente código:

```java
public class Pedido {
    private double monto;
    private String cliente;

    public Pedido(double monto, String cliente) {
        this.monto = monto;
        this.cliente = cliente;
    }

    public void imprimirFactura() {
        System.out.println("Factura para: " + cliente);
        System.out.println("Monto: $" + monto);
    }

    public void aplicarDescuento(double porcentaje) {
        this.monto -= this.monto * (porcentaje / 100);
    }
}
```

**a)** Identifique si el diseño viola algún principio de diseño. Justifique.
**b)** Proponga un diseño alternativo respetando los principios SRP y OCP.

a) Se viola el OCP y el SRP ya que una sola clase (pedido) esta teniendo mas de una tarea, y además en el caso de querer modificar algunos criterios de descuentos, se debería modificar toda la clase.
b) Se debería separar las clases en pedido y factura, y que los criterios de descuento sean manejados (por qué no) en factura como un método o subclase clase anónima.

## Ejercicio 4: Programación Concurrente en Java

1. Analice el siguiente programa en Java:

```java
public class CuentaBancaria {
    private int balance = 1000;

    public synchronized void retirar(int monto, String usuario) {
        if (balance >= monto) {
            System.out.println(usuario + " está retirando: $" + monto);
            balance -= monto;
            System.out.println("Balance restante: $" + balance);
        } else {
            System.out.println(usuario + " intentó retirar $" + monto + ", pero no hay fondos suficientes.");
        }
    }

    public int getBalance() {
        return balance;
    }
}

public class Main {
    public static void main(String[] args) {
        CuentaBancaria cuenta = new CuentaBancaria();

        Thread usuario1 = new Thread(() -> cuenta.retirar(700, "Usuario1"));
        Thread usuario2 = new Thread(() -> cuenta.retirar(500, "Usuario2"));

        usuario1.start();
        usuario2.start();
    }
}
```

**a)** Explique qué podría suceder si se elimina la palabra clave `synchronized` del método `retirar`.
**b)** Si ambos usuarios intentan retirar dinero simultáneamente, ¿qué salidas posibles podrían observarse? Justifique.

Tenemos un código, donde el main esta recibiendo dos hilos con tareas de manera concurrente y de manera independiente entre cada una. A estos se les pasa por parámetro un nombre y un balance en cuenta. Esto pasará a la clase CuentaBancaria, que tiene definida un balance estándar para cada una, y de esta manera cada hilo accedes a esta función y verifica si puede o no retirar dinero y su balance. El synchronized esta permitiendo que estos hilos accedan a la ejecución y funcione de manera sincronizada justamente, para evitar que ambos hilos intenten acceder al mismo tiempo a la tarea y se produzca una condición de carrera e incluso incongruencias en los balances. Incluso si cambiamos el orden de los hilos pero mantenemos el Synchronized, no afectará nada ya que este orden solo espera en ese lugar a que la tarea termine su ejecución y pase a la otra. Pero sin esta palabra, se produciría la sobreescritura y condición de carrera.

## Ejercicio 5: Programación Orientada a Objetos (POO)

Se tiene el siguiente programa en Java:

```java
abstract class Dispositivo {
    private String nombre;

    public Dispositivo(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }

    public abstract void calcularConsumo();
}

class DispositivoFijo extends Dispositivo {
    private double consumoPorHora;

    public DispositivoFijo(String nombre, double consumoPorHora) {
        super(nombre);
        this.consumoPorHora = consumoPorHora;
    }

    @Override
    public void calcularConsumo() {
        System.out.println("Consumo por hora de " + getNombre() + ": " + consumoPorHora + " kWh.");
    }
}

class DispositivoVariable extends Dispositivo {
    private int horasUso;
    private double consumoBase;

    public DispositivoVariable(String nombre, int horasUso, double consumoBase) {
        super(nombre);
        this.horasUso = horasUso;
        this.consumoBase = consumoBase;
    }
    @Override
    public void calcularConsumo() {
        System.out.println("Consumo total de " + getNombre() + ": " + (horasUso * consumoBase) + " kWh.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dispositivo lampara = new DispositivoFijo("Lámpara", 0.5);
        Dispositivo aireAcondicionado = new DispositivoVariable("Aire Acondicionado", 8, 1.2);

        lampara.calcularConsumo();
        aireAcondicionado.calcularConsumo();
    }
}
```

**a)** Analice el diseño del programa y explique el propósito de las clases.

**b)** Indique si el programa cumple con los principios OCP (Open/Closed Principle) y LSP (Liskov Substitution Principle). Justifique.

**c)** Realice un diagrama UML representando las clases del programa y sus relaciones.

Ejercicio 6.a) Se tiene un programa que instancia desde el main dos métodos lampara y aireAcondicionado, de las clases dispositivoFijo, y dispositivoVariable respectivamente, a las cuales se les pasa por parámetro un string que indica el tipo de dispositivo, y sus horas de consumo o kw (dependiendo del tipo de dispositivo). Estos son evaluados por la clase Dispositivo, que se encargar de identificar los dispositivos y calcula el consumo basado en las propiedades específicas de cada subclase. El propósito de la clase abstracta `Dispositivo` es servir como una plantilla común para dispositivos, mientras que las subclases `DispositivoFijo` y `DispositivoVariable` implementan cálculos específicos según su naturaleza.

b) No, no cumple con OCP ya que por ejemplo, si quisiéramos cambiar el cálculo de las tasas de consumo de un dispositivo, estaríamos modificando directamente una clase y su ciclo. Sin embargo no hay ningún indicio de que viole LSP ya que las subclases cumplen con la funcionalidad esperada de la clase base Dispositivo.

## Ejercicio 6: Cálculo Lambda

Considere la siguiente expresión lambda:

**((λx.(λy.x+y)) 3 5)**

**a) Reduzca la expresión paso a paso hasta llegar a su forma normal.**

**b) Explique el resultado obtenido y su interpretación**.

((λx.(λy.x+y)) 3 5)

Sustituimos x por 3: (λy.3+y) 5

Sustituimos y por 5: 3 + 5

Resultado final reducido: 8

## Ejercicio 7: Programación Funcional en Clojure

Dado el siguiente código en Clojure:

```
(defn fibonacci [n]
  (if (<= n 1)
    n
    (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))

(defn parallel-fibonacci [nums]
  (let [results (atom [])]
    (doseq [n nums]
      (future (swap! results conj (fibonacci n))))
    @results))
```

**a) Explique el propósito de las funciones `fibonacci` y `parallel-fibonacci`.**

**b) ¿Qué ocurriría si la función `parallel-fibonacci` recibe una lista con valores muy grandes, como `[35 36 37]`? Analice su impacto en la concurrencia y el rendimiento.**

Ejercicio 9.a) Se tienen dos funciones, fibonacci que calcula el n-ésimo número de Fibonacci., a los cuales mientras el n no sea menor a uno, se le suma los dos últimos n generados, tales que quede una secuencia ejemplo de 0, 1, 1, 2, 3, 5, 8 , .... Por otra parte tenemos la comparación de la secuencia fibonacci generada (parallel-fibonacci) que almacena esta secuencia y usa `future` para calcular números de Fibonacci en paralelo y acumular los resultados en un `atom`.

b) Si solo hablamos de rendimiento y eficiencia temporal, el hecho de no establecer ningún rango o filtro a este mapeo de la secuencia de n elementos generados, hace que la búsqueda sea infinita y no tenga un parámetro para almacenar, y mucho menos justamente con el mapeo y escalado de números altos como 35 o 36 o 37, y sus secuencias de fibonacci. Sumado a esto, a pesar del uso de swap!, no hay nada que evite problemas en la concurrencia ya que la secuencia de n es infinita y no termina nunca de acceder al valor a mutar, sumado a que no puede almacenar en ningún lado estos.

## Ejercicio 8: Programación Concurrente en Java

Considere el siguiente programa en Java:

```java
class Contador {
    private int contador;

    public synchronized void incrementar() {
        contador++;
    }

    public synchronized void decrementar() {
        contador--;
    }

    public int getContador() {
        return contador;
    }
}

public class Main {
    public static void main(String[] args) {
        Contador contador = new Contador();

        Thread hilo1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                contador.incrementar();
            }
        });

        Thread hilo2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                contador.decrementar();
            }
        });

        hilo1.start();
        hilo2.start();

        try {
            hilo1.join();
            hilo2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Valor final del contador: " + contador.getContador());
    }
}
```

**a) Explique el propósito del programa y analice su salida esperada.**

**b) ¿Qué problemas podrían surgir si se elimina la palabra clave `synchronized` de los métodos `incrementar` y `decrementar`?**

Ejercicio 10.a) Tenemos una clase main, que recibe dos tareas de dos hilos (1 y 2) de manera concurrente e independientes entre sí. Por otra parte tenemos a la clase contador, que recibe como parámetro un int contador que inicia desde 0 e incrementa de a uno. Cada hilo incrementa o decrementa el mismo contador compartido, y el valor final dependerá del orden en que los hilos ejecuten sus operaciones. La salida esperada es `0`, ya que ambos hilos realizan 1000 operaciones opuestas en el mismo contador.De esta manera, los dos hilos acceden cada uno a su respectivo funcionamiento, y tanto en el start() como en el join() de cada hilo, se esperará que empiece como termine la ejecución de cada hilo para poder luego ver el resultado de cada uno. Como resultado, veremos o 1000 para el hilo 1, o 0 para el hilo 2 (con sus respectivas secuencias) como valor final del contador para cada uno.

b) El synchronized evita condiciones de carrera en este programa, ya que permite que ambos hilos accedan de manera independiente y sincronizada a la función. Sin embargo, al estar especificado qué función debe tomar cada hilo, no estoy seguro si esto afectará totalmente al acceso y ejecución correcta de cada una de las tareas al final. Esto puede llevar a resultados impredecibles, como un valor final incorrecto o incluso excepciones debido a la falta de atomicidad.