Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza

4 de septiembre de 2021

Examen de **Tecnología de Programación**

Grado en Ingeniería Informática

Duración total del examen: **2 horas 30 minutos**

NOTA RECORDATORIA. La evaluación final consta de las siguientes pruebas:

Prueba escrita → 60 % de la nota (mínimo 5 para aprobar)
Prácticas y Problemas → 40 % de la nota

Ejercicio 1 [2 puntos]

Existen ciertos comportamientos de programación funcional que pueden ser representados en lenguajes orientados a objetos.
Elije un lenguaje orientado a objetos y define los siguientes elementos:

1. Una función unaria que, dado un parámetro real (`float`), devuelva su **valor absoluto** (`float`) que se pueda pasar como parámetro.
2. Una función unaria que, dado un parámetro real (`float`), devuelva un booleano que indica si el valor está en el rango $[0 \ldots 1]$.
3. Una función equivalente a la función de Haskell
   ```
   map :: (a -> b) -> [a] -> [b]
   ```
   que dada una función unaria y una colección, devuelva una colección nueva con el resultado de aplicar la función unaria a los elementos de la colección de entrada en la misma posición.
4. Una función equivalente a la función de Haskell
   ```
   all :: (a -> Bool) -> [a] -> Bool
   ```
   que dada una función booleana (predicado), y una colección, devuelve true si todos los elementos de la colección cumplen el predicado.
5. Utilizando todas las funciones anteriores, define una función '`unit`', que dada una colección, devuelve un valor booleano que indica si la colección está dentro del cubo unidad, es decir, todos los valores están en el rango $[-1 \ldots +1]$.

No es obligatorio que reproduzcas el comportamiento funcional en general: no es necesario que las funciones se implementen mediante recursividad ni que se reproduzcan otras características de los lenguajes funcionales como el *currying*.

Además de lo que se te pide, puedes definir otras clases, funciones y métodos si lo consideras necesario.

Puedes utilizar cualquier elemento de la biblioteca estándar del lenguaje elegido.

Para representar las colecciones, puedes elegir una colección concreta como `std::vecto` (en C++) o `ArrayList` (en Java).

Ejercicio 2 [2 puntos]

Muchos lenguajes de programación implementan alguna forma de **programación genérica**, lo que permite **polimorfismo paramétrico**. A continuación verás tres ejemplos de código similares de diferentes lenguajes en los que se muestran funciones (o métodos) genéricos junto con sus respectivos programas principales que los utilizan.

C++

```
#include <iostream>
using namespace std;

template<typename T>
T max(T a, T b) {
  if (a>b) return a;
  else return b;
}

int main() {
  cout<<max(3,5)<<'\n';
}
```

Java

```
class Max {
 public static <T> T
   max(T a, T b) {
     if (a>b) return a;
     else return b;
 }

 public static void
   main(String[] args[]) {
     System.out.println(max(3,5));
 }
}
```

Haskell

```
max :: t -> t -> t
max a b
  | a > b     = a
  | otherwise = b


main = print (max 3 5)
```

Sobre estos ejemplos realiza las siguientes tareas:

1. **Responde:** De los tres ejemplos, sólo compila correctamente el de uno de los lenguajes. ¿Cuál es? ¿Por qué siendo ejemplos tan similares sólo compila ese? ¿Por qué no compilan los otros dos? Asume que la función o método `max` no está ya definida en ninguno de los lenguajes, y que el error de compilación tiene que ver con la programación genérica y no con otros elementos sintáticos del lenguaje.
2. Elije uno de los dos lenguajes en los que el ejemplo no compila y **diseña una modificación** (lo más pequeña posible) que habría que aplicar al ejemplo correspondiente para que compilase correctamente. Justifica por qué dicha modificación hace que el ejemplo compile.
3. Sobre el ejemplo que compila, **añade** una única línea de código al programa principal que utilice la función o método genérico y que provoque que el programa no compile. Explica por qué el ejemplo no compila al añadir esa línea.
4. **Explica** el concepto de **restricciones de tipos** en programación genérica y cómo se ha visto reflejado en tus respuestas anteriores.

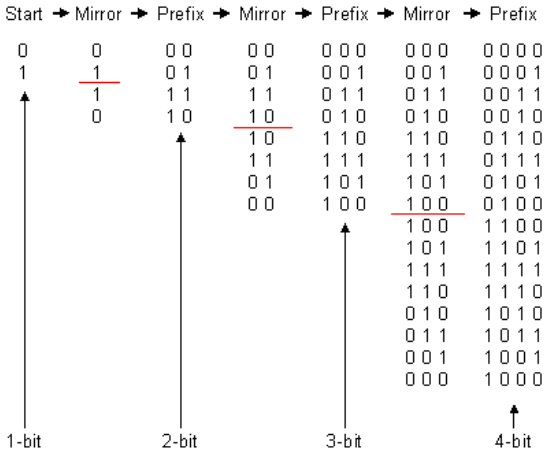
Ejercicio 3 [3 puntos]

El **Código Gray** es una forma de representar números en binario de forma que entre dos números consecutivos sólo cambia un bit.

Decimal Values	Natural Binary Code	Gray Code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Si los representamos como cadenas de 0's y 1's, los códigos de Gray para los números de n bits pueden obtenerse recursivamente. Por ejemplo, para 3 bits los obtenemos a partir de los de 2 bits:

códigos de Gray para números de 2 bits: ["00", "01", "11", "10"]
lista invertida: ["10", "11", "01", "00"]
lista original prefijada con ceros: ["000", "001", "011", "010"]
lista invertida prefijada con unos: ["110", "111", "101", "100"]
concatenación de ambas: ["000", "001", "011", "010", "110", "111", "101", "100"]



Se pide: implementar en Haskell la función

```
gray :: Int -> [String]
```

que devuelve una lista de cadenas con los códigos de Gray de los números de n-bits

```
> gray 1
["0","1"]

> gray 2
["00","01","11","10"]

> gray 3
["000", "001", "011", "010", "110", "111", "101", "100"]
```

NOTA: Puedes utilizar todas las funciones de la biblioteca estándar de Haskell que consideres necesarias.

Ejercicio 4 [3 puntos]

Para lugares con conectividad limitada, se desea implementar el sofware de un cajero automático que no trabaje directamente contra la base de datos del banco, sino que vaya almacenando las operaciones que se realizan en una estructura de datos en memoria que por la noche se vuelca a la central.

Cada mañana el cajero descarga la información de las cuentas (identificador y saldo) que ira usando durante el dia.

Las operaciones posibles son ingresos, retiradas de efectivo y transferencias, y su informacion asociada depende de cada una: siempre existe una cantidad de dinero asociada, y puede existir un identificador de cuenta origen (en las retiradas), destino (en los ingresos) o ambas (en las transferencias). El cajero almacena una única estructura de datos (registro) con todas las operaciones que se van realizando durante el día.

En el lenguaje orientado a objetos que elijas (C++ o Java), define una serie de clases para representar la información almacenada en el cajero, que implemente las siguientes clases y métodos:

- **Cuenta**: almacena su identificador (entero) y su saldo en euros (real).
- **Ingreso**, **Extraccion** y **Transferencia**, para representar las **operaciones**, con los identificadores de cuenta necesarios y la cantidad.
- **Cajero**: almacena la lista de Cuentas y ofrece los métodos necesarios para realizar las operaciones, con los parámetros necesarios:

  ```
  ... ingreso( ... )
  ... retirada( ... )
  ... transferencia( ... )
  ```

  Dichos metodos realizan la operación correspondiente sobre la(s) cuenta(s) y guardan en una estructura de datos (registro, también almacenado en el Cajero) la operación correspondiente.
  Ten en cuenta que la operación puede no ser posible (si no hay saldo), por lo que esos métodos pueden fallar.

Para realizar ciertas comprobaciones, al final de cada día se necesita saber la lista de cuentas que han realizado alguna operación en el cajero. Implementa un método

```
... cuentasUsadas( ... )
```

que devuelva la lista de los identificadores de las cuentas que han realizado alguna operación en el cajero.

Además de los que te pedimos explícitamente, puedes definir todos los métodos y clases que creas necesarios.

Se valorará que no exista código repetido o métodos innecesarios, dependiendo de las carácterísticas de cada clase.
Puedes utilizar toda la funcionalidad definida en las bibliotecas estándar de C++ o Java.