Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza
1542

Examen de Teoría

30 de mayo de 2023

Tecnología de Programación
Grado en Ingeniería Informática
Duración: 2 horas 30 minutos

Recuerda:

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

El examen se realiza de forma **INDIVIDUAL**.

Entrega:

- Escribe tu nombre completo en **todas** las hojas del examen.
- Responde a cada ejercicio comenzando **en una hoja diferente**, se entregan por separado.
- No se puede entregar la resolución escrita a **lápiz**.

Ejercicio 1 **[2 puntos]**

En una biblioteca de cálculo científico necesitamos almacenar valores numéricos, de forma que:

- los números racionales se representan de forma exacta, con su numerador y denominador.
- el resto (irracionales) se representan mediante un valor real (que es aproximado).
- para ambos debe ser posible obtener su valor como un número real.
- las estructuras de datos (vectores, listas, matrices, etc) deben poder almacenar números de ambos tipos mezclados.

Queremos que nuestra biblioteca se pueda utilizar desde C++ y Haskell.

Define en ambos lenguajes un tipo de datos **Number**, que permita almacenar números racionales e irracionales como se ha definido arriba.

Define en ambos lenguajes una función **`value(...)`** que nos devuelva el valor real de un número, sea del tipo que sea.

¿ Que funcionalidades del lenguaje empleas en cada uno de ellos (C++ y Haskell) ?

Puedes definir los tipos y funciones/métodos adicionales que consideres oportuno.

(continúa en la siguiente página...)

Ejercicio 2 **[3 puntos]**

Para realizar **estudios genealógicos** se dispone (en Haskell) de todas las relaciones entre padre/madre e hijo/hija en una lista de tuplas con sus nombres: `[(String,String)]`, donde en cada tupla el primer elemento es el nombre del progenitor, y el segundo el de su descendiente directo (hijo/a). A partir de esta información, se pretenden encontrar a todas las personas que cumplan una relación de parentesco con una persona dada:

- Hermano: dos personas que comparten todos sus padres y madres.
- Descendiente: hijo o hija, y recursivamente sus respectivos hijo o hijas (nietos, bisnietos, tataranietos...) sin límite de número de generaciones de diferencia.

Se pide que implementes en Haskell las dos funciones siguientes:

```haskell
encontrarHermanos :: [(String,String)] -> String -> [String]
encontrarDescendientes :: [(String, String)] -> String -> [String]
```

donde el primer argumento son todas las relaciones entre padre/madre (primer elemento de cada tupla) e hijo/hija (segundo elemento de cada tupla), el segundo argumento es la persona de la que se están buscando los parientes, y el dato devuelto es una lista con los nombres de todas las personas que cumplen dicho parentesco con la persona dada. La primera de las funciones devuelve los nombres de todos los hermanos y la segunda de todos los descendientes. No se deben devolver nombres repetidos.

Ejemplo de ejecución:

```haskell
> familia :: [(String,String)]
> familia = [("Juan Carlos","Elena"), ("Juan Carlos","Felipe"), ("Juan Carlos","Cristina"), ("Sofia","Elena"),("Sofia","Felipe"),
("Sofia","Cristina"),("Inaki","Juan U."), ("Inaki","Pablo"),("Inaki","Miguel"),("Inaki","Irene"),("Cristina","Juan U."),
("Cristina","Pablo"),("Cristina","Miguel"),("Cristina","Irene"),("Elena","Felipe Juan Froilan"),("Elena","Victoria Federica"),
("Jaime","Felipe Juan Froilan"),("Jaime","Victoria Federica"),("Juan","Juan Carlos"),("Juan","Pilar"),("Juan","Alfonso"),
("Juan","Margarita"),("Maria","Juan Carlos"),("Maria","Pilar"),("Maria","Alfonso"),("Maria","Margarita"),("Pilar","Simoneta"),
("Pilar","Bruno"),("Pilar","Beltran"),("Pilar","Juan G."),("Pilar","Fernando"),("Luis","Simoneta"),("Luis","Bruno"),
("Luis","Beltran"),("Luis","Juan G."),("Luis","Fernando"),("Margarita","Alfonso Juan"),("Margarita","Maria Sofia"),
("Carlos","Alfonso Juan"),("Carlos","Maria Sofia")]
```

```haskell
> encontrarHermanos familia "Juan Carlos"
["Alfonso","Pilar","Margarita"]
> encontrarDescendientes familia "Juan Carlos"
["Elena","Felipe","Cristina","Felipe Juan Froilan","Victoria Federica","Juan U.","Pablo","Miguel","Irene"]
```

Notas:

- Aunque se pudiera formular como un árbol, no se te está pidiendo que crees un árbol en ningún momento.
- Los nombres representados en `String` son únicos, no hay dos personas con exactamente el mismo nombre.
- Puedes asumir que los datos que aparecen en la lista de tuplas `[(String,String)]` son correctos y bien estructurados, sin nadie que sea su propio abuelo (como pasaba en Futurama), así que no tienes que comprobar potenciales recursividades infinitas.
- Puedes añadir otras funciones además de la que se te pide, pero para cada función que añadas deberás también incluir tanto la cabecera con los tipos de datos involucrados como un comentario que explique qué es lo que hace.

Puedes utilizar las funciones de la biblioteca estándar de Haskell que consideres oportuno.
Se valorará la concisión, eficiencia y brevedad de la solución propuesta.

(continúa en la siguiente página...)

Ejercicio 3 **[5 puntos]**

Para el desarrollo de un juego de rol queremos representar a nuestros personajes mediante una biblioteca de clases. Elige un lenguaje orientado a objetos (C++ o Java), e implementa con él lo que se pide en este ejercicio.

En el mundo de nuestro juego existen personajes de distintas razas (por simplificar, de momento lo dejaremos en humanos, elfos y enanos), que pueden tomar el papel de distintos tipos de personajes (digamos exploradores, guerreros y magos). En dicho mundo se desarrolla una lucha entre dos bandos que no tienen ninguna preferencia por raza o tipo, asi que en ambos existen personajes de todas las razas y tipos.

Durante la lucha, ambos bandos tienen patrullas (formadas por una serie de personajes variados) viajando por el mundo, que, desgraciadamente, cuando se encuentran acaban combatiendo.

Las caracteristicas de los personajes son las siguientes:

- Todos tienen un nombre (cadena de caracteres) y un rango (entero).
- Todos tienen un capacidad de 'defensa' a los ataques, pero es distinta para cada raza (definido mediante un valor entero), aunque no se puede modificar para un personaje en particular.
- Los guerreros tienen una capacidad de ataque físico ('fuerza', valor entero) que no tienen los demás.
- Los magos tienen una capacidad de ataque mágico ('magia', valor entero) que no tienen los demás.
- Una patrulla es un grupo de personajes, y tiene un poder de 'ataque' (suma de la 'fuerza' y la 'magia' de sus miembros), y una capacidad de 'defensa' (el total de la de sus miembros).

En el lenguaje elegido (C++ o Java), implementa lo siguiente:

- Define los tipos necesarios para representar tanto las razas como los tipos generales de personajes, de forma que permitan definir de forma fácil personajes como un MagoEnano o un EploradorElfo.
  Se valorará la duplicación mínima de código y la extensibilidad del mismo, de forma que añadir una nueva raza (p.e. trolls) o un nuevo tipo (p.e. clérigos), implique modificaciones mínimas.
- Define los tipos necesarios para implementar una patrulla.
- Implementa un método `'capitan()'` en la patrulla que devuelva el miembro de mayor rango.
- Implementa un método `'atacantes()'` en la patrulla que devuelva la lista de sus miembros con poderes de ataque (fuerza o magia).
- Implementa un método `'vence(...)'` en la patrulla, que devuelva un booleano que representa si la patrulla vence a otra, a la que se considera que ataca. Cuando una patrulla ataca a otra, vence si su capacidad de 'ataque' supera a la de 'defensa' de la otra.

Puedes utlizar todas las clases y herramientas definidas en la STL de C++ o la JFC de Java.