Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza
1542

Examen de Teoría

7 de junio de 2022

Tecnología de Programación
Grado en Ingeniería Informática
Duración: 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]**

A continuación tienes dos programas, escritos en C++ y en Haskell, que realizan la misma función:

C++

```
float step(int n,float f,float l)
{
  if (n==0)
    return 0.0;
  else
    return (l-f)/float(n-1);
}

vector<float> u {1.0,3.0,5.0};
vector<float> v;

int main() {
  cout << step(u.size(),u.front(),u.back())
       << endl;
  cout << step(v.size(),v.front(),v.back())
       << endl;

  return 0;
}
```

Haskell

```
step :: Int -> Float -> Float -> Float
step n f l
  | n == 0    = 0
  | otherwise = (l-f)/(fromIntegral (n-1))



u = [1.0,3.0,5.0]
v = []

main = do
  print $ step (length u) (head u) (last u)
  print $ step (length v) (head v) (last v)
```

1. ¿ Compilan los dos programas ?
   Puedes suponer que tanto en C++ como en Haskell se han incluido los ficheros de cabecera o módulos necesarios.
2. ¿ Se ejecutan correctamente los dos programas ?
   ¿ Cuál es la salida por pantalla en cada caso ?
3. Si hay diferencias en su ejecución, ¿ por qué se producen esas diferencias ?
4. ¿ Qué concepto/propiedad de los dos lenguajes es responsable de ese comportamiento ?

Define ese concepto, y la forma en que está implementado en cada uno de los dos lenguajes.

Ejercicio 2 **[2 puntos]**

Dentro de la Programación Orientada a Objetos, el uso de **programación genérica** permite diseñar código dejando un elemento del lenguaje (habitualmente, un tipo de datos) de forma paramétrica, de tal forma que dicho elemento puede ser concretado a posteriori. Esto permite lo que se denomina **polimorfismo paramétrico**.
Sobre este tema realiza las siguientes tareas:

1. En el lenguaje orientado a objetos de tu elección (C++ o Java) escribe un **pequeño ejemplo de código** que incluya:
   - Una clase genérica (que dependa de un parámetro tipo) suficientemente simple como para resolver el resto de tareas expuestas a continuación.
   - Una función (o método de otra clase) genérico que utilice de alguna manera esa clase genérica y que dependa únicamente del mismo parámetro tipo que la clase.
   - Un programa principal que, como mínimo, invoque a la función anterior tres veces, instanciándola para tres tipos de datos diferentes, uno por invocación. De las tres invocaciones de la función, una de ellas deberá concretar un tipo de datos que provoque un error de compilación gracias al control de tipos del lenguaje. Las otras dos deberán compilar correctamente.
2. Sobre tu ejemplo anterior, **explica**:
   - En qué parte del código se aprecian las dos fases de la programación genérica: definición e instanciación.
   - En qué parte del código se puede apreciar el polimorfismo paramétrico.
3. Explica por qué da un error de compilación en la invocación concreta de la función genérica y razona que habría que hacer para que esa misma llamada no diese un error de compilación para el mismo tipo de datos.

Ejercicio 3 **[3 puntos]**

Mediante un sistema de aprendizaje automático, se está entrenando la IA de un dron para realizar misiones de reconocimiento sobre un terreno rectangular. El terreno se divide en una serie de $m \times n$ **sectores** rectangulares de tamaño uniforme, indexados bidimensionalmente, que se representan mediante tuplas de dos números enteros que representan la posición de dicho sector. El sector de una de las esquinas es el `(1,1)` y el sector de la esquina opuesta el `(m,n)`, siendo `m` y `n` el número de sectores en cada una de las dimensiones del rectángulo, respectivamente.

Para entrenar la IA, se necesita, entre otras cosas, comprobar que la trayectoria seguida es **óptima**.

Se pide que desarrolles en Haskell una función `trayectoriaEsOptima` que haga esta comprobación, con la siguiente definición:

```haskell
trayectoriaEsOptima :: (Int,Int) -> [(Int,Int)] -> Bool
```

donde el primer argumento es una tupla que contiene las dimensiones `(m,n)` del terreno (en número de sectores) y el segundo argumento es una trayectoria representada mediante una lista con todos los sectores que recorrería el dron, en orden de recorrido.

Se dice que una trayectoria es óptima si cumple con lo siguiente:

- La trayectoria seguida por el dron no sale a ningún sector que esté fuera del terreno rectangular (su posición nunca es menor que 1 y mayor que el tamaño del terreno en la dimensión correspondiente).
- La trayectoria no contiene saltos, es decir, cualesquiera dos sectores consecutivos de la trayectoria son contiguos. Se considera que dos sectores son contiguos si colindan horizontalmente, verticalmente o diagonalmente. Es decir, excepto en los bordes del terreno cada sector se considera que tiene 8 vecinos (sectores contiguos).
- La trayectoria recorre todos los sectores del terreno, desde el `(1,1)` hasta el `(m,n)`, una única vez. Es decir: no hay sectores por los que el dron pase dos o más veces y tampoco hay sectores del terreno que se queden sin explorar.

Ejemplo de ejecución:

```haskell
ghci> trayectoriaEsOptima (2,3) [(1,1),(1,2),(1,3),(2,3),(2,2),(2,1)]
True  -- Correcto, la trayectoria cumple todas las condiciones

ghci> trayectoriaEsOptima (2,3) [(1,1),(1,2),(1,3),(2,3),(2,2),(2,1),(1,1)]
False -- No es optima porque repite el sector (1,1)

ghci> trayectoriaEsOptima (2,3) [(1,1),(1,2),(1,3),(2,3),(2,2),(2,1),(3,1)]
False -- No es optima porque el sector (3,1) esta fuera del terreno

ghci> trayectoriaEsOptima (2,3) [(1,1),(1,3),(1,2),(2,3),(2,2),(2,1)]
False -- No es optima porque el sector (1,1) y el sector (1,3) no son contiguos.
```

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

Ejercicio 4 **[3 puntos]**

Para el control de acceso a ciertos edificios e instalaciones se está intentando desarrollar un sistema basado en tarjetas inteligentes, de tal forma que el poseedor de cada tarjeta tiene acceso a unas instalaciones específicas en base a una serie de permisos.

Se pide que, sobre un lenguaje orientado a objetos (C++ o Java, a tu elección), implementes el software de gestión de permisos de acceso, bajo las siguientes premisas:

- Cada **tarjeta** se identifica con un código único (entero) y tiene información de su poseedor (NIF y nombre completo).
- Cada **tarjeta** puede pertenecer a varios **grupos**. Un grupo es un conjunto de tarjetas que disponen de unos permisos en común, y que se identifica mediante un código único (entero). La única información que se necesita del grupo es ese código, y no es necesario almacenar para un grupo todas las tarjetas que pertenecen al mismo, aunque una tarjeta sí que debe tener la información de a qué grupos pertenece.
- Cada **instalación** se identifica con un código único (entero) y tiene información de su nombre.
- Cada **instalación** puede pertenecer a varias **categorías**. Una categoría es un conjunto de instalaciones que disponen de unos permisos en común, y que se identifica mediante un código único (entero). La única información que se necesita de una categoría es ese código, y no es necesario, para una categoría, almacenar todas las instalaciones que pertenecen a la misma, aunque una instalación sí que debe tener la información de a qué categorías pertenece.
- Un **permiso** da acceso a una tarjeta o un grupo, a una instalación o una catergoría de instalaciones.
  - Los permisos pueden ser **individuales** (dan acceso a una única tarjeta, identificada mediante su código) o bien **de grupo** (dan acceso a cualquier tarjeta que perteneza a un grupo determinado).
  - El acceso proporcionado por el permiso puede ser para una única **instalación** o para una **categoría** de instalaciones.
- Existe un único **conjunto de permisos** para todas las tarjetas, grupos, instalaciones y categorías que se pretenden representar.

Define una biblioteca de clases que permita representar todos estos conceptos. Para probar su funcionamiento, implementa un método en aquella clase que represente el **conjunto de permisos** que, dada una **tarjeta** y una **instalación** (clases completas, no sólo sus identificadores), indique mediante un booleano si dicha tarjeta tiene o no acceso a dicha instalación. Se considera que una tarjeta puede acceder a una instalación si existe uno o más permisos que le proporcionan dicho acceso, ya sea individual o de grupo, para una instalación concreta o para una categoría a la que pertenezca la instalación correspondiente.

Se valorará la no duplicidad del código, y que no exista código o métodos innecesarios, dependiendo de las características de cada clase. También se valorará la escalabilidad de tu solución (la facilidad con la que se pueden añadir permisos que hagan otras comprobaciones más sofisticadas, como por ejemplo que los lunes sólo pueden entrar los pertenecientes a un grupo que además tengan un código par en su tarjeta). Puedes utilizar toda la funcionalidad definida en las bibliotecas estándar del lenguaje elegido.