Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza

11 de junio 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:

(a) Dos funciones binarias de dos parámetros reales (`float`) que devuelvan un número real (`float`), y que se puedan pasar como parámetro. Una de ellas representará la **suma** y la otra el **producto** de dos números reales.

(b) Una función equivalente a la función de Haskell
`zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
que dada una función binaria y dos colecciones, devuelva una colección nueva con el resultado de aplicar la función binaria a los elementos de las colecciones de entrada en las mismas posiciones.

(c) Una función equivalente a la función de Haskell
`foldl :: (a -> b -> a) -> a -> [b] -> a`
que dada una función binaria, un elemento y una colección, aplica la función al elemento y al primer elemento de la colección, el resultado al segundo elemento de la lista y así sucesivamente hasta finalizar la colección, devolviendo el resultado.

(d) Utilizando todas las funciones anteriores, define una función nueva `dot` que devuelva el producto escalar de dos vectores ($\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$) representados mediante colecciones.

Puedes utilizar cualquier elemento de la biblioteca estándar del lenguaje elegido.
Además de lo que se te pide, puedes definir otras clases, funciones y métodos si lo consideras necesario.
No es obligatorio que reproduzcas el comportamiento funcional en general: no es necesario que las funciones se implemente mediante recursividad ni que se reproduzcan otras características de los lenguajes funcionales como el *currying*.
Para representar las colecciones, puedes elegir una colección concreta como `std::vector` (en C++) o `ArrayList` (en Java).

Ejercicio 2 [2 puntos]

Dentro de la Programación Orientada a Objetos, el establecimiento de una relación de **herencia** entre dos clases desencadena una serie de mecanismos que permiten **polimorfismo por inclusión**.
Sobre este tema realiza las siguientes tareas:

(a) En el lenguaje orientado a objetos de tu elección (C++ o Java) escribe un **pequeño ejemplo de código** que incluya:

- Dos clases no abstractas relacionadas mediante herencia que permitan polimorfismo por inclusión, estableciendo comportamientos diferentes de un mismo método.
- Una función (o método estático de otra clase) no genérico que permita como parámetro las dos clases anteriores y que invoque al menos a un método que pueda tener diferentes comportamientos.
- Un programa principal que invoque a la función correspondiente con objetos pertenecientes a las dos clases.

(b) Explica dónde se ve en tu ejemplo anterior el polimorfismo por inclusión y qué mecanismos de la herencia lo están permitiendo.

(c) Explica qué tipo de asociación de métodos (estática o dinámica) está siendo aplicado por el lenguaje y cómo cambiaría el resultado de tu programa si la asociación de métodos fuera la contraria.

Ejercicio 3 [3 puntos]

La **integración numérica** incluye una amplia gama de algoritmos para calcular el valor numérico de una integral definida. Ejemplos de métodos sencillos de integración numérica son los siguientes:

Regla del Trapecio
$$\int_a^b f(x)\ dx \approx \hat{T}(f,a,b) = \frac{b-a}{2}\left(f(a)+f(b)\right) \tag{1}$$

Regla de Simpson
$$\int_a^b f(x)\ dx \approx \hat{S}(f,a,b) = \frac{b-a}{6}\left(f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right) \tag{2}$$

La Regla de Simpson es un poco más precisa porque tiene en cuenta el valor en el punto central del intervalo, aunque si el intervalo es lo suficientemente lineal ("plano"), los dos valores son casi iguales. Sin embargo estos métodos son bastante inexactos cuando la función $f(x)$ varía mucho. En esos casos se suele subdividir el intervalo de integración $[a,b]$ de alguna forma, y se evalúa la integral a trozos.

Las **reglas compuestas** subdividen el rango de integración $[a,b]$ en un número n de subintervalos regulares $[a_i,b_i] = [a+i*dx, a+(i+1)*dx]$, con $dx=(b-a)/n$, para $i \in [0,n-1]$ y aplican una regla simple en cada uno de los intervalos, sumando los resultados. Conforme el número n se hace más grande, la regla es más precisa pero también requiere un tiempo de cálculo superior.

Las **reglas adaptativas** o anidadas, consisten en, dada una tolerancia t, se estima el error del cáculo como la diferencia de los valores de dos reglas (una de mayor precisión $\hat{S}$ y otra de menor precisión $\hat{T}$) para el intervalo donde se evalua la función ($|\hat{S}-\hat{T}|$), y si ese error es menor que la tolerancia, el resultado es $\hat{S}$ (que es más precisa), y en caso contrario se subdivide recursivamente el rango de integración $[a,b]$ por la mitad y se evalúa la integral en cada una de las dos mitades:

$$\int_a^b f(x)\ dx \approx \begin{cases} \int_a^c f(x)\ dx + \int_c^b f(x)\ dx, \text{ donde } c=(a+b)/2 & \text{si } |\hat{S}(f,a,b)-\hat{T}(f,a,b)| > t \\ \hat{S}(f,a,b) & \text{en caso contrario} \end{cases} \tag{3}$$

Menores valores de tolerancia t implican mayor precisión pero también mayor tiempo de cálculo.

Se pide definir en Haskell una serie de funciones:

(a) Un par de funciones `trapecio` y `simpson`, que a partir de una función $f(x)$ y una tupla que define el intervalo (a,b), calculen el valor de la integral con cada una de las reglas simples, respectivamente. Se tienen que poder invocar de la siguiente forma:

```
trapecio (\x -> 1.0/(x*x)) (0.01,1.0)
simpson (\x -> 1.0/(x*x)) (0.01,1.0)
```

(b) `integralCompuesta`, que dada una función que evalúa un método de integración (como las dos anteriores u otra con el mismo interfaz), un número de intervalos n, una función de un parámetro $f(x)$ y un rango de integración representado mediante una tupla (a,b), calcule numéricamente la integral compuesta de varios pasos del método de integración, invocable así:

```
integralCompuesta simpson 10 (\x -> 1.0/(x*x)) (0.01,1.0)
```

(c) `integralAdaptativa`, que dadas dos funciones que evalúan métodos de integración (más preciso y menos preciso), una tolerancia t, una función de un parámetro $f(x)$ y un rango de integración representado mediante una tupla (a,b), calcule numéricamente la integral mediante la regla adaptativa como se describe en (3), invocable así:

```
integralAdaptativa simpson trapecio 0.001 (\x -> 1.0/(x*x)) (0.01,1.0)
```

Puedes asumir que las funciones devuelven números reales `Double` y que la tolerancia y los rangos también son números reales `Double`. Se valorará la concisión y brevedad de la solución propuesta.

Ejercicio 4 [3 puntos]

Una empresa de reciclado de residuos desea informatizar la gestión de los mismos.
Elige un lenguaje orientado a objetos (C++ o Java) para implementar lo que se pide en este ejercicio.

La gestión de residuos presenta las siguientes particularidades:

- Para cualquier **Residuo** se conoce la información (el NIF) del particular o empresa que lo ha generado y al que se le recoge.
- Los residuos pueden clasificarse como **Orgánicos** o **Inorganicos**. La diferencia es que los orgánicos requieren ser procesados antes de un tiempo máximo de almacenamiento (definido en días), dependiente de cada tipo (mucho mas corto por ejemplo en la basura que en el aceite), y los inorgánicos no.
- También pueden clasificarse como **Sólidos** o **Líquidos**, lo que influye en el medio de transporte elegido para recogerlos. Los sólidos se caracterizan por su peso, y los líquidos por su volumen.

Los residuos concretos con los que trabajará el programa pueden ser por ejemplo aceite (orgánico y líquido), pintura (inorgánico y líquido), basura (orgánico y sólido), vidrio (inorgánico y sólido), etc...

Define una biblioteca de clases para representar la información de los tipos de residuos finales.

Para optimizar la **Recogida**, el programa almacena una clase con el conjunto de residuos pendientes de recoger. Esa clase debe ofrecer además la siguiente funcionalidad:

- Dado el NIF de un particular/empresa, calcular el volumen total de los residuos líquidos que hay que recoger en ella, para elegir un camión de la capacidad adecuada.
- Calcular la lista de empresas (NIFs) que tienen pendientes de recoger residuos orgánicos antes de un número de días dado, para ajustarse al tiempo de procesado de los mismos (supondremos que se procesan el mismo día de la recogida).

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