Trabajo Práctico - Programación Funcional

Gestionando Viajes

Introducción a la Programación - Segundo cuatrimestre de 2024

Fecha límite de entrega: Martes 08 de Octubre - 23:59 hs

Introducción

El objetivo de este Trabajo Práctico es aplicar los conceptos de programación funcional vistos en la materia para resolver ejercicios utilizando el lenguaje de programación Haskell.

La reconocida agencia de viajes online *Aterrizar.com* nos contactó para ayudarlos a resolver ciertos problemas relacionados con la oferta de vuelos entre distintas ciudades. Nos han brindado información sobre las ciudades que conectan y la duración entre sus vuelos. El sistema requiere verificar la validez de las ofertas de vuelo, determinar posibles rutas, modernizar la flota para reducir tiempos de vuelo, y encontrar la ciudad con mayor conectividad, entre otros aspectos.

Para ello se deben implementar una serie de ejercicios respetando su especificación, además de otros criterios que se detallan en las pautas de entrega.

Pautas de Entrega

Para la entrega del trabajo práctico se deben tener en cuenta las siguientes consideraciones:

- El trabajo se debe realizar en grupos de tres estudiantes *de forma obligatoria*.
  No se aceptarán trabajos de menos ni más integrantes.
  Ver aviso en el campus de la materia donde indica el link para registrar los grupos.
- Se debe implementar un conjunto de casos de test para todos los ejercicios (no es obligatorio para las funciones auxiliares que definan).
- Los programas deben pasar con éxito los casos de test entregados por la cátedra (en el archivo test-catedra.hs), sus propios tests, y un conjunto de casos de test "secreto".
- El archivo con el código fuente debe tener nombre `Solucion.hs`. Además, en el archivo entregado debe indicarse, en un comentario arriba de todo: nombre de grupo, nombre, email y DNI de cada integrante. En caso de que algún integrante haya abandonado la materia, aclararlo entre paréntesis. Recordar que no se acepta un número de integrantes diferente a tres, pero aceptamos que queden menos en caso de que alguien abandone.
- El código debe poder ser ejecutado en el GHCI instalado en los laboratorios del DC, sin ningún paquete especial.
- No está permitido alterar los nombres de las funciones a implementar ni los tipos de datos. Deben mantenerse tal cual descargan el template.
- Pueden definir todas las funciones auxiliares que se requieran.
- No está permitido utilizar técnicas no vistas en clase para resolver los ejercicios (como por ejemplo, alto orden). En el campus, pueden encontrar un listado de todas las funciones válidas a usar. Si usan funciones fuera de esa lista en algún ejercicio, se desaprobará el TP.

Se evaluarán las siguientes características:

- **Correctitud**: todos los ejercicios deben estar bien resueltos, es decir, deben respetar su especificación.
- **Declaratividad**: los nombres de las funciones que se definan, así como los nombres de las variables, deben ser apropiados.
- **Consistencia**: el código debe atenerse al uso correcto de las técnicas vistas en clase como recursión o *pattern matching*.
- **Prolijidad**: evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (por el enunciado o por ustedes mismos). En caso de funciones complejas, analizar si es conveniente realizar comentarios sobre el código.
- **Testing**: Todos los ejercicios del enunciado deben tener sus propios casos de test que pasen correctamente. Un ejercicio sin casos de test se considera no resuelto. Los casos de test deben cubrir distintos escenarios de uso de las funciones.

Método de entrega

Se espera que el trabajo grupal lo realicen de forma incremental utilizando un sistema de control de versiones, como Git. Para la entrega, sólo vamos a trabajar con GitLab (pueden ser `git.exactas.uba.ar` si tienen usuario, o deben crearse un usuario en `gitlab.com`). El repositorio debe ser privado, deben estar todos los integrantes del grupo y, además, deben agregar con rol "Developer" un usuario de los docentes:

- Si usan `git.exactas.uba.ar`: Deben agregar al usuario *docentes.ip*
- Si usan `gitlab.com`: Deben agregar al usuario *ip.dc.uba*

La entrega debe ser realizada ***únicamente*** por un integrante del grupo. Debe realizarlo quien haya anotado al grupo en el link para registrar los grupos. La entrega consta de la URL del repositorio y un commit particular. Los docentes descargarán el código para ese commit. En caso que el link o el commit no sean válidos, el trabajo estará desaprobado.

Antes de enviar el trabajo, se recomienda fuertemente:

- Clonar el repositorio con el link y commit proporcionados. Chequear que ambos son válidos.
- Ejecutar los casos de test de la cátedra y sus propios casos de test, sobre el repositorio recién clonado.
- Todos los casos de test deben pasar satisfactoriamente, y el archivo con la implementación debe poder cargarse sin generar errores.

Tipos de datos

Para especificar los problemas que debemos resolver, usaremos los siguientes renombres de tipos:

- Renombre Ciudad = String
- Renombre Duracion = Float
- Renombre Vuelo = Ciudad × Ciudad × Duracion
- Renombre AgenciaDeViajes = $seq\langle Vuelo \rangle$

En Haskell los implementaremos de la siguiente manera:

- `type Ciudad = String`
- `type Duracion = Float`
- `type Vuelo = (Ciudad, Ciudad, Duracion)`
- `type AgenciaDeViajes = [Vuelo]`

Especificación

A continuación se especifican todos los ejercicios que se deben programar en Haskell.

Ejercicio 1

`problema vuelosValidos` (vuelos: AgenciaDeViajes) : Bool {
`requiere:` {True}
`asegura:` {res = true↔ (no hay elementos repetidos en *vuelos*, y además no hay dos elementos *v1* y *v2* en *vuelos* tal que $(v1 \neq v2 \land v1_1 = v2_1 \land v1_2 = v2_2)$ y además para todo elemento v de *vuelos* se cumple *vueloValido(v)*)}
}

Nota: Recordar que si tenemos, por ejemplo, la tupla $A = (5, 7)$ y queremos referirnos al primer componente, usamos la notación A_1.

`problema vueloValido` (vuelo: Vuelo) : Bool {
`requiere:` {True}
`asegura:` {res = true↔ La duración del *vuelo* es mayor estricto a 0, y el origen y destino de *vuelo* son diferentes}
}

Ejemplo:

```
entrada: vuelos = [("BsAs", "Rosario", 5.0)]
res: True
```

Ejercicio 2

`problema ciudadesConectadas` (agencia: AgenciaDeViajes, ciudad: Ciudad) : $seq\langle Ciudad \rangle$ {
`requiere:` {vuelosValidos(*agencia*)}
`asegura:` {*res* no tiene elementos repetidos}
`asegura:` {*res* contiene todas las ciudades a las que se puede llegar con un vuelo desde *ciudad* teniendo en cuenta los vuelos ofrecidos por *agencia*}
`asegura:` {*res* contiene todas las ciudades a las que se puede llegar con un vuelo hacia *ciudad* teniendo en cuenta los vuelos ofrecidos por *agencia*}
`asegura:` {*res* no contiene ninguna ciudad que no esté conectada mediante vuelos directos con *ciudad*, teniendo en cuenta los vuelos ofrecidos por *agencia*}
}

Ejemplo:

```
entrada: agencia = [("BsAs", "Rosario", 5.0)]
         ciudad = "Rosario"
res: ["BsAs"]
```

Ejercicio 3

`problema modernizarFlota` (agencia: AgenciaDeViajes) : AgenciaDeViajes {

`requiere:` {vuelosValidos(*agencia*)}

`asegura:` {*res* contiene la misma cantidad de vuelos que *agencia*}

`asegura:` {Si dos ciudades c_1 y c_2 estaban conectadas mediante un vuelo en *agencia*, entonces *res* contiene un vuelo conectando esas dos ciudades (manteniendo cuál es el origen y cuál el destino). Además, en *res*, ese vuelo demora un 10 % menos del tiempo que demoraba el vuelo entre esas ciudades en *agencia*}

}

Ejemplo:

```
entrada: agencia = [("BsAs", "Rosario", 10.0)]
res: [("BsAs", "Rosario", 9.0)]
```

Ejercicio 4

`problema ciudadMasConectada` (agencia: AgenciaDeViajes) : Ciudad {

`requiere:` {vuelosValidos(*agencia*)}

`requiere:` {$|agencia| > 0$}

`asegura:` {*res* es alguna de las ciudades con más conexiones teniendo en cuenta los vuelos ofrecidos por *agencia*}

}

Las conexiones de una ciudad c pueden pensarse como $|ciudadesConectadas(agencia, c)|$.

Ejemplo:

```
entrada: agencia = [("BsAs", "Rosario", 10.0), ("Rosario", "Córdoba", 7.0)]
res: "Rosario"
```

Ejercicio 5

`problema sePuedeLlegar` (agencia: AgenciaDeViajes, origen: Ciudad, destino: Ciudad) : Bool {

`requiere:` {vuelosValidos(*agencia*)}

`asegura:` {*res* = true$\leftrightarrow$ existe al menos una ruta, directa o con a lo sumo una escala, desde *origen* a *destino* en los vuelos ofrecidos por *agencia*}

}

Ejemplo:

```
entrada: agencia = [("BsAs", "Rosario", 5.0), ("Rosario", "Córdoba", 5.0),
                    ("Córdoba", "BsAs", 8.0)]
         origen = "BsAs"
         destino = "Córdoba"
res: True
```

Ejercicio 6

`problema duracionDelCaminoMasRapido` (agencia: AgenciaDeViajes, origen: Ciudad, destino: Ciudad) : Duracion {

`requiere:` {vuelosValidos(*agencia*)}

`requiere:` {Existe al menos una ruta, ya sea directa o con a lo sumo una escala, desde *origen* a *destino* a partir de los vuelos ofrecidos por *agencia*}

`asegura:` {*res* es el camino más corto (en cuanto a duraciones) para llegar desde *origen* a *destino* en los vuelos ofrecidos por *agencia*}

}

Ejemplo:

```
entrada: agencia = [("BsAs", "Rosario", 5.0), ("Rosario", "Córdoba", 5.0),
                    ("Córdoba", "BsAs", 8.0)]
         origen = "BsAs"
         destino = "Córdoba"
res: 10.0
```

Ejercicio 7

`problema puedoVolverAOrigen` (agencia: AgenciaDeViajes, origen: Ciudad) : Bool {

`requiere:` {vuelosValidos(*agencia*)}

`asegura:` {*res* = true↔ existe al menos una ruta (directa o con una o más escalas), que permita partir y volver a la misma ciudad *origen* teniendo en cuenta los vuelos ofrecidos por *agencia*}

}

Ejemplo:

```
entrada: agencia = [("BsAs", "Rosario", 5.0), ("Rosario", "Córdoba", 5.0),
                    ("Córdoba", "BsAs", 8.0)]
         origen = "BsAs"
res: True
```