Trabajo Práctico 1
Programación Funcional

Paradigmas de Lenguajes de Programación — 1^{er} cuat. 2020

Fecha de entrega: 7 de mayo de 2020

1. Introducción

La princesa Mérida[1] quiere salir a investigar las luces mágicas pero su madre, la reina Elinor[2] no le deja divertirse hasta no haber cumplido con todas sus labores de princesa. Las tareas que tiene Mérida asignadas hoy no son difíciles, pero son muy repetitivas y la tediosa rutina la está agotando.

Para poder terminar cuanto antes, Mérida busca ayuda para automatizar su trabajo. Su primo lejano Hipo le recomendó usar un nuevo lenguaje imperativo súper eficiente. Pero al no poder expresar correctamente sus ideas, Mérida decidió pasarse al paradigma funcional.

Algunas tareas son muy simples. Estas son las *tareas básicas*. Otras tareas son más *complejas* y pueden descomponerse en subtareas. Cada tarea compleja está compuesta de dos subtareas (cada una de estas, puede ser a su vez otra tarea compleja). Las subtareas que componen a una tarea compleja pueden tener dependencias entre sí. Si ese es el caso, deben realizarse una después de la otra. Además, puede ser que haya que esperar un tiempo entre que finalizó la primera y se puede empezar con la segunda. Si las dos subtareas que componen a una tarea compleja no tienen dependencias entre sí, entonces pueden ser realizadas en cualquier orden.

Las tareas básicas son atómicas. Una vez que se comienza una no se puede empezar con otra hasta no haber terminado la primera. Durante el intervalo de espera entre dos tareas dependientes se puede avanzar con otra tarea siempre que esta última no dependa de otra que no haya sido realizada antes.

Las tareas pueden ser representadas en Haskell de la siguiente forma:

```
data Tarea =
    Basica String Int |
    Independientes Tarea Tarea |
    DependeDe Tarea Tarea Int
```

donde:

- `Basica n h` representa una tarea básica llamada `n` que requiere `h` horas para ser realizada.
- `Independientes t1 t2` representa una tarea compleja compuesta por las tareas `t1` y `t2` que son independientes entre sí.
- `DependeDe t1 t2 h` representa una tarea compleja compuesta por la tarea `t1` que depende de la tarea `t2` y requiere un intervalo de `h` horas entre la finalización de la segunda y el comienzo de la primera.

2. Resolver

Ejercicio 1

Definir y dar el tipo de las siguientes funciones. Por tratarse de esquemas de recursión, para definir estas funciones se permite utilizar **recursión explícita**.

[1] https://disney.fandom.com/es/wiki/M%C3%A9rida
[2] https://disney.fandom.com/es/wiki/La_Reina_Elinor

(a) `recTarea`, que representa el esquema de recursión primitiva *recr* sobre `Tarea`.

(b) `foldTarea`, que representa el esquema de recursión estructural *foldr* sobre `Tarea`. Notar que puede definirse en términos de `recTarea`.

Ejercicio 2

El primer paso para encarar cualquier trabajo es estar consciente de la magnitud del mismo. Definir y dar el tipo de las siguientes funciones.

(a) `cantidadDeTareasBasicas`, que dada una lista de tareas devuelve la cantidad de tareas básicas en las que se pueden descomponer las tareas originales.

(b) `cantidadMaximaDeHoras`, que dada una lista de tareas devuelve la máxima cantidad de horas[3] que se requieren para completar todas las tareas. Considerar que dos tareas cualesquiera de la lista original son independientes entre sí.

(c) `tareasMasLargas`, que dada una cantidad de horas `h` y una lista de tareas devuelve otra lista que contiene sólo las tareas de la lista original[4] cuya duración total puede llegar a ser mayor que `h`.

Ejercicio 3

Mérida descubre que una lista con dos tareas es el equivalente a una tarea compuesta de dos subtareas independientes. Para simplificar la representación, decide deshacerse de las listas. Definir y dar el tipo de la función `chauListas` que dada una lista **no vacía** de tareas devuelve una única tarea compleja compuesta de todas las tareas de la lista original. No es importante la forma en que se agrupen los pares de subtareas en tareas complejas.

Ejercicio 4

Mérida está considerando varias alternativas para empezar a realizar las tareas que tiene pendientes. Una opción es empezar por las más cortas para que la sensación de progreso la incentive a continuar. Otra opción es empezar por las más largas y dejar las más cortas para cuando esté más cansada.

Definir y dar el tipo de las siguientes funciones.

(a) `tareasBasicas`, que dada una tarea devuelve una lista (sin orden establecido) con todas las tareas básicas en las que se puede descomponer la tarea original.

(b) `esSubTareaDe`, que dado un nombre `n` y una tarea `t` determina si la tarea `t` tiene alguna subtarea básica con nombre `n`.

(c) `tareasBasicasIniciales`, que dada una tarea `t` devuelve una lista con las subtareas básicas de `t` que no tienen ninguna dependencia. Es decir, las tareas que podrían ser la primera tarea a realizar.

(d) `tareasBasicasQueDependenDe`, que dado un nombre `n` y una tarea `t` devuelve una lista con las subtareas básicas de `t` que dependen (ya sea directa o indirectamente) de haber realizado antes una tarea con nombre `n`.

[3] Es decir, si no se aprovechan los intervalos de espera entre dos tareas dependientes para adelantar una tercera tarea

[4] Es decir, no se deben agregar subtareas como tareas a la lista.

Ejercicio 5

Al final Mérida decidió que lo mejor es empezar con la tarea básica con mayor cantidad de dependencias. Definir y dar el tipo de la función `cuelloDeBotella` que dada una tarea devuelve el nombre de la tarea básica para la cual existen más tareas básicas que dependen de ella. En caso de empate se puede devolver el nombre de cualquiera de ellas. Pista: usar `sortBy`[5]

Ejercicio 6

Gracias al poder de la programación funcional, Mérida pudo terminar todas sus tareas temprano y ahora tiene mucho tiempo para ir a explorar. Sin embargo, quedó tan manija que ahora no puede evitar modelar todo lo que encuentra en el paradigma funcional. Al encontrar las luces mágicas, descubrió que cada luz mágica se puede representar en Haskell como una función que dado un elemento de un tipo devuelve otro elemento del mismo tipo.

```
type LuzMagica a = (a → a)
```

Las luces mágicas están dispuestas en una fila y para encontrar su destino, Mérida debe partir de un punto inicial `zi` y llegar a un punto final `zf`. La forma de alcanzar su destino `zf` es ir aplicando sobre `zi` las sucesivas funciones que le provée cada una de las luces mágicas. A Mérida le gustaría saber qué tan difícil es alcanzar su destino así que quiere escribir una función que le indique la cantidad de luces mágicas que tiene que recorrer.

Definir y dar el tipo de la función `pasos` que dado un elemento de cualquier tipo `zf` y una lista de luces mágicas del mismo tipo devuelva una función que dado un elemento `zi` (del mismo tipo que `zf`) devuelva la cantidad de pasos necesaria para llegar de `zi` a `zf`, es decir, la cantidad de funciones de la lista tales que, al aplicarlas en orden sobre `zi` se obtiene `zf`. Tener en cuenta que, como las luces mágicas son mágicas, la lista es infinita y, por lo tanto, la función puede colgarse si no hay un número `n` tal que al aplicarle a `zi` las primeras `n` funciones se obtenga `zf`.

Formalmente, si las luces mágicas son las funciones ℓ_i y existe un k que es el mínimo natural que cumple $\ell_k(\ell_{k-1}(...\ell_1(z_i))) = z_f$, entonces

pasos z_f $[\ell_1,\ \ell_2,\ \ell_3,\ ...]$ $=$ f_{z_f} tal que $f_{z_f}(zi)$ $=$ k, o lo que es equivalente,

pasos z_f $[\ell_1,\ \ell_2,\ \ell_3,\ ...]$ z_i $=$ k

[5] http://zvon.org/other/haskell/Outputlist/sortBy_f.html

Tests

Parte de la evaluación de este Trabajo Práctico es la realización de tests. Tanto HUnit[6] como HSpec[7] permiten hacerlo con facilidad.

En el archivo de esqueleto que proveemos se encuentran tests básicos utilizando *HUnit*. Para correrlos, ejecutar dentro de *ghci*:

```
> :l tp1.hs
[1 of 1] Compiling Main             ( tp1.hs, interpreted )
Ok, modules loaded: Main.
> main
```

Para instalar HUnit usar: `> cabal install hunit` o bien `apt install libghc-hunit-dev`.
Para instalar cabal ver: https://wiki.haskell.org/Cabal-Install

Pautas de Entrega

Se debe entregar el código impreso con la implementación de las funciones pedidas. Cada función asociada a los ejercicios debe contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el código fuente en Haskell a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-PF]` seguido inmediatamente del **nombre del grupo**.
- El código Haskell debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto (puede adjuntarse un `.zip` o `.tar.gz`).
- El código entregado **debe** incluir tests que permitan probar las funciones definidas.

El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté **adecuadamente** comentado (son comentarios adecuados los que ayudan a entender lo que no es evidente o explican decisiones tomadas; no son adecuadas las traducciones al castellano del código). Los objetivos a evaluar son:

- Corrección.
- Declaratividad.
- Prolijidad: evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso adecuado de funciones de alto orden, currificación y esquemas de recursión: es necesario para los ejercicios que usen las funciones que vimos en clase y aprovecharlas, por ejemplo, usar zip, map, filter, take, takeWhile, dropWhile, foldr, foldl, listas por comprensión, etc, cuando sea necesario y no volver a implementarlas.

Salvo donde se indique lo contrario, **no se permite utilizar recursión explícita**, dado que la idea del TP es aprender a aprovechar lo ya definido. Se permite utilizar listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `Data.Char`, `Data.Function`, `Data.List`, `Data.Maybe`, `Data.Ord` y `Data.Tuple`. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas. Pueden escribirse todas las funciones auxiliares que se requieran, pero estas no pueden usar recursión explícita (ni mutua, ni simulada con `fix`).

Importante: se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

[6] https://hackage.haskell.org/package/HUnit
[7] https://hackage.haskell.org/package/hspec

Referencias del lenguaje Haskell

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la última versión del lenguaje Haskell a la fecha, disponible online en: http://www.haskell.org/onlinereport/haskell2010.
- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en: http://learnyouahaskell.com/chapters.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en http://book.realworldhaskell.org/read.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos *parciales*, online en http://www.haskell.org/hoogle.
- **Hayoo!**: buscador de módulos no estándar (i.e. aquellos no necesariamente incluídos con la plataforma Haskell, sino a través de **Hackage**), online en http://holumbus.fh-wedel.de/hayoo/hayoo.html.