Trabajo Práctico 1
Programación Funcional

Naves Espaciales

Paradigmas de Lenguajes de Programación
1er cuatrimestre, 2017

Fecha de entrega: Martes 18 de Abril

Introducción

Construcción de naves espaciales

Las grandes construcciones espaciales son ensambladas en el espacio mediante el acoplamiento de distintos módulos, que son puestos en órbita de forma independiente, dado que es inviable poner en órbita la construcción completa.

Considere la siguiente representación para naves espaciales, donde dos naves pueden fusionarse a través de un módulo conector para formar una nave más grande y potente:

```
data Componente = Contenedor | Motor | Escudo | Cañón
data NaveEspacial = Módulo Componente NaveEspacial NaveEspacial | Base Componente
```

Distinguimos al nodo raíz donde se tripula la nave y consideraremos los siguientes tipos de componentes:

- Contenedor (C), cada uno aporta una unidad de almacenamiento.
- Motor (M) propulsa la nave, cada motor ubicado en un nodo hoja provee una unidad de empuje.
- Escudos de fuerza (E) que protegen a la nave de colisiones con objetos pequeños.
- Cañones de plasma (P) que sirven para evitar colisiones con objetos grandes convirtiéndolos en pequeños.

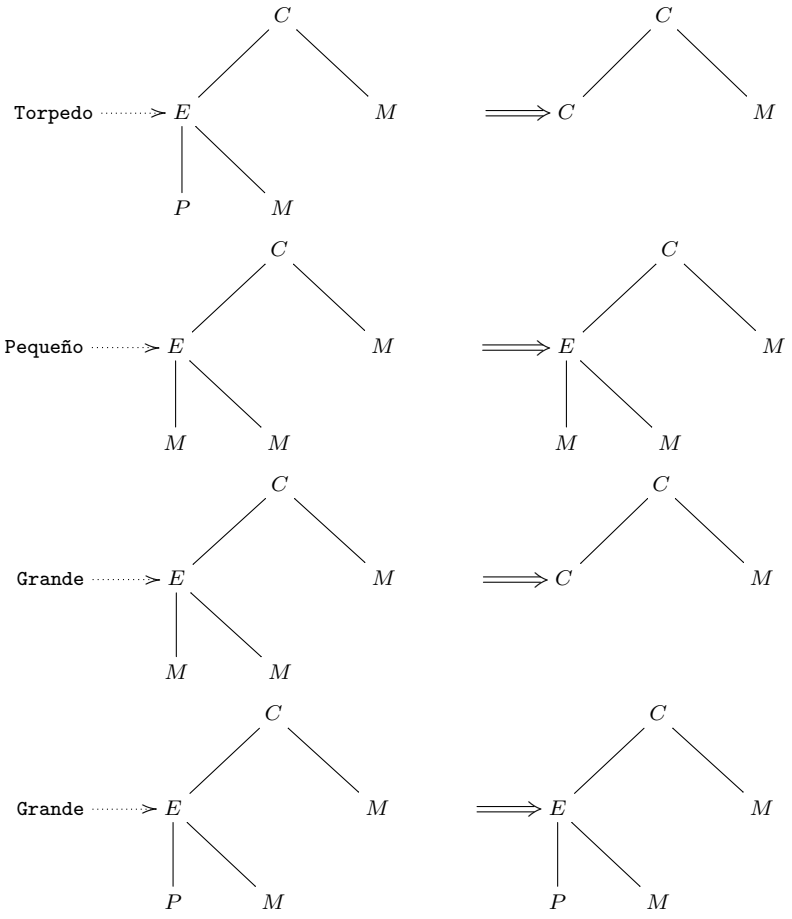
Peligros

Consideraremos que los peligros encontrados en el espacio pueden provenir de la izquierda o la derecha de la cabina, pero nunca del frente o de atrás, por lo que los modelaremos de la siguiente forma:

```
data Dirección = Babor | Estribor
data TipoPeligro = Pequeño | Grande | Torpedo
type Peligro = (Dirección, Int, TipoPeligro)
```

El entero asociado a un peligro indica a qué nivel con respecto a la cabina impactará un objeto. Identificamos tres tipos de peligros: objetos pequeños, grandes y torpedos. Estos últimos no pueden ser evadidos por las naves, por lo que siempre producen un impacto. Si un objeto pequeño impacta sobre un escudo, no produce daño. Si el objeto es grande y el componente impactado está protegido por un cañón, entonces se transforma en un objeto pequeño. Se considera que un componente está protegido por un cañón cuando la (sub)nave que tiene como raíz a ese componente contiene un cañón. Si el objeto es un torpedo no se puede evitar el impacto. En otras palabras, todos los peligros producen daño a menos que se trate de un objeto pequeño que impacta contra un escudo, o un objeto grande que impacta contra un escudo protegido por un cañón.

Ejemplo 1



Las naves del ejemplo reciben un impacto de distintos objetos por la izquierda (babor) a nivel 1 (la raíz es el nivel 0).

Si el peligro no es neutralizado, todos los componentes dependientes del nodo impactado se desprenden y el nodo impactado se convierte en un contenedor (espacio libre).

Los peligros solo pueden impactar contra el nodo más externo del nivel correspondiente (es decir, el de más a la izquierda si están a babor, o el de más a la derecha si se encuentran a estribor).

Implementación

A continuación se detallan los problemas a resolver.

Esquemas de recursión

Ejercicio 1

Implementar el esquema de recursión estructural sobre las naves espaciales (foldNave) y dar su tipo. Por ser este un esquema de recursión, se permite utilizar recursión explícita para definirlo.

Operaciones sobre naves

Implementar las siguientes funciones (puede ser necesario crear una o más abstracciones adicionales):

Ejercicio 2

1. `capacidad :: NaveEspacial -> Int`, que retorna la capacidad de almacenamiento de la nave, donde cada contenedor aporta una unidad a esta capacidad.
2. `poderDeAtaque :: NaveEspacial -> Int`, que retorna la cantidad de cañones que tiene la nave.
3. `puedeVolar :: NaveEspacial -> Bool`, que indica si la nave contiene al menos un motor.
4. `mismoPotencial :: NaveEspacial -> NaveEspacial -> Bool`, indica si las dos naves tienen los mismos componentes, en iguales cantidades.

Ejercicio 3

`mayorCapacidad :: [NaveEspacial] -> NaveEspacial`, que dada una lista no vacía de naves retorna la que tiene la máxima capacidad de almacenamiento.

Ejercicio 4

`transformar :: (Componente -> Componente) -> NaveEspacial -> NaveEspacial` que, dada una función que reemplaza componentes, devuelve la nave espacial transformada mediante los reemplazos correspondientes.

Enfrentando peligros

Ejercicio 5

Definir la función `impactar :: Peligro -> NaveEspacial -> NaveEspacial`, que devuelve la nave resultante luego de enfrentar el peligro correspondiente (según se detalló en la sección **"Peligros"**).

Para este ejercicio puede utilizarse recursión explícita. Se debe explicar en un comentario por qué el esquema foldNave no es adecuado para esta función.

Ejercicio 6

Implementar la función `maniobrar :: NaveEspacial -> [Peligro] -> NaveEspacial`, que devuelve el resultado de maniobrar una nave a través de una serie de peligros en orden secuencial de izquierda a derecha.

Ejercicio 7

`pruebaDeFuego :: [Peligro] -> [NaveEspacial] -> [NaveEspacial]`,
que dada una lista de peligros y una lista de naves retorna la lista de naves que pueden sobrevivir a los peligros (i.e. las que tendrían motores funcionales luego de enfrentar los peligros sucesivos).

Las difíciles

Ejercicio 8

Implementar las siguientes funciones:

1. `componentesPorNivel :: NaveEspacial -> Int -> Int`, que dadas una nave y un nivel indica cuántos componentes contiene la nave en el nivel indicado, viendo a la nave como un árbol cuya raíz (cabina) es el nivel 0. **Ayuda:** pensar cuál es el verdadero tipo de esta función.

2. `dimensiones :: NaveEspacial -> (Int, Int)`,

   que dada una nave retorna su largo y ancho (pensando el largo como la cantidad de componentes de la rama más larga, y ancho como la cantidad de componentes del nivel más ancho).

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 las características enumeradas en el ítem anterior. Se permite utilizar listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `List`, `Maybe`, `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.

Tests: se recomienda la codificación de tests. Tanto HUnit https://hackage.haskell.org/package/HUnit como HSpec https://hackage.haskell.org/package/hspec permiten hacerlo con facilidad.

Para instalar HUnit usar: `> cabal install hunit`

Para instalar cabal ver: https://wiki.haskell.org/Cabal-Install

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. aquéllos no necesariamente incluídos con la plataforma Haskell, sino a través de **Hackage**), online en http://holumbus.fh-wedel.de/hayoo/hayoo.html.