Taller de Programación Funcional

Árboles (de verdad)

Paradigmas de Lenguajes de Programación
2do cuatrimestre, 2018

Fecha de entrega: 11 de Septiembre del 2018

Introducción

Seguramente, a lo largo de la carrera habrán trabajado numerosas veces con árboles, pero casi con seguridad, nunca los habrán usado para modelar árboles. En este taller saldaremos esta deuda.

Construcción de árboles

Los árboles se construyen a partir de brotes, que son las partes capaces de sostener un componente básico (madera, hojas, flores o frutos) y las ramas, que unen dos subárboles con un componente entre ellos dos.

Considere la siguiente representación para árboles:

```
data Componente = Madera | Hoja | Fruto | Flor

data Arbol = Rama Componente Arbol Arbol | Brote Componente
```

Consideraremos los siguientes tipos de componentes:

- Madera (M), cada uno aporta una unidad de peso.
- Hoja (H) produce la fotosíntesis, cada hoja ubicada en la punta de una rama (es decir, no en el interior del árbol, pues no recibiría luz) aporta una unidad de energía.
- Fruto (Fr) son el objetivo de los animales atacantes.
- Flores (Fl) que sirven para distraer atacantes.

Asaltos

Consideraremos que los animales pueden acceder al árbol desde la izquierda o la derecha , pero nunca del frente o de atrás, por lo que los modelaremos de la siguiente forma:

```
data Dirección = Izquierda | Derecha
data TipoHambre = Gula | Hambre | Inanición
type Animal = (Int, Dirección, TipoHambre)
```

Los animales se acercan al árbol porque tienen hambre y quieren comer sus frutos. El entero asociado a un animal indica en qué nivel con respecto a la raíz alcanzará al árbol (los animales pueden saltar desde abajo del árbol o venir volando y posarse en alguna rama).

Además, identificamos tres tipos de hambre: gula, hambre e inanición. Si un animal goloso alcanza un fruto, le da un pequeño mordisco pero no produce un daño al árbol. Si el animal está hambriento, pero el componente alcanzado está protegido por una flor, entonces el animal se distrae y se convierte en goloso. Se considera que un componente está protegido por una flor cuando el (sub)árbol que parte de ese componente contiene una flor. Si el animal está en estado de inanición está descontrolado y no se puede evitar el daño. En otras palabras, todos los animales producen daño a menos que se trate de un animal goloso en un fruto, o un animal hambriento que alcanza un fruto protegido por una flor.

Ejemplo 1

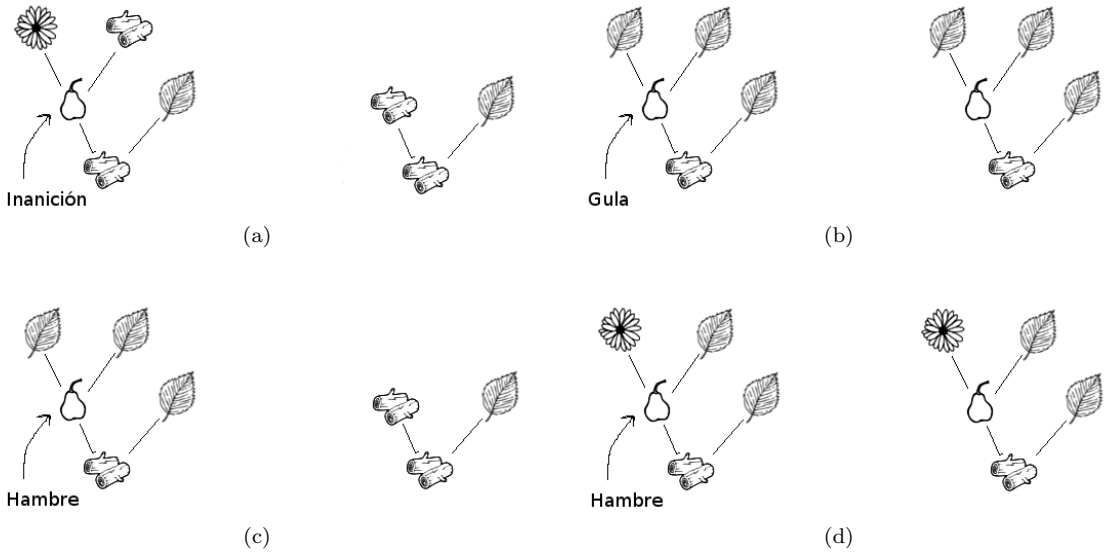

Figura 1: Antes y después de que un animal pasa por un árbol.

Las árboles del ejemplo reciben a los distintos animales por la izquierda a nivel 1 (la raíz es el nivel 0).

Si el asalto produce daño, todos los componentes dependientes del nodo impactado se desprenden y el nodo impactado se convierte en madera.

Los animales solo pueden alcanzar el nodo más externo del nivel correspondiente (es decir, el de más a la izquierda si viene por dirección izquierda, o el de más a la derecha si viene por dirección derecha).

Implementación

A continuación se detallan los problemas a resolver.

Esquemas de recursión

Ejercicio 1

Implementar el esquema de recursión estructural sobre los árboles (foldArbol) y dar su tipo. Por ser este un esquema de recursión, se permite utilizar recursión explícita para definirlo.

Operaciones sobre árboles

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

Ejercicio 2

1. `peso :: Arbol -> Int`, que retorna el peso del árbol, donde únicamente la madera aporta una unidad al peso.
2. `perfume :: Arbol -> Int`, que retorna la cantidad de flores que tiene la árbol.
3. `puedeVivir :: Arbol -> Bool`, que indica si el árbol tiene por lo menos una hoja para hacer fotosíntesis.
4. `mismosComponentes :: Arbol -> Arbol -> Bool`, indica si los dos árboles tienen los mismos componentes en iguales cantidades.

Ejercicio 3

`masPesado :: [Arbol] -> Arbol`, que dada una lista no vacía de árboles retorna el que tiene mayor peso.

Ejercicio 4

`crecer :: (Componente -> Componente) -> Arbol -> Arbol` que,
dada una función que reemplaza componentes, devuelve el árbol transformado mediante los reemplazos correspondientes.

`ultimaPrimavera :: Arbol -> Arbol` que devuelve un arbol con flores en vez de hojas (observemos que el árbol así florecido no puede vivir).

Vienen a comer

Ejercicio 5

Definir la función `comer :: Animal -> Arbol -> Arbol`, que devuelve el árbol resultante luego de ser asaltado por el animal correspondiente (según se detalló en la sección **"Asaltos"**).

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

Ejercicio 6

Implementar la función `alimentar :: Arbol -> [Animal] -> Arbol`,
que devuelve el resultado de hacer que arbol alimente a una serie de animales, en el orden en que aparecen en la lista.

Ejercicio 7

`sobrevivientes :: [Animal] -> [Arbol] -> [Arbol]`,
que dada una lista de animales y una lista de árboles retorna la lista de árboles que pueden sobrevivir a la secuencia de animales (i.e. los que tendrán hojas luego de enfrentar los asaltos en forma **sucesiva**).

Las difíciles

Ejercicio 8

Implementar las siguientes funciones:

1. `componentesPorNivel :: Arbol -> Int -> Int`, que dadas un árbol y un nivel indica cuántos componentes contiene el árbol en el nivel indicado, teniendo en cuenta que la raíz es el nivel 0. Por ejemplo, todo árbol tendrá un componente en el nivel 0, 0 o 2 componentes en el nivel 1, a lo sumo 4 en el nivel 2, etc. **Ayuda:** pensar cuál es el verdadero tipo de esta función.

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

   que dado un árbol retorna su alto y se ancho (pensando el alto como la cantidad de componentes de la rama más larga desde la raíz, 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). Las correcciones que haremos harán foco en:

- 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 taller 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`).

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.