Trabajo Práctico 1
Programación Funcional

Paradigmas de Lenguajes de Programación — 1^{er} cuat. 2019

Fecha de entrega: 23 de abril

1. Introducción

En este trabajo práctico trabajaremos con árboles binarios definidos en Haskell de la siguiente manera:

```
data AB a = Nil | Bin (AB a) a (AB a)
```

Donde `Nil` representa el árbol vacío y `Bin i v d` representa un árbol binario cuya raíz tiene el valor `v` de algún tipo `a` y sus subárboles izquierdos y derechos son `i` y `d` respectivamente.

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**.

- `recAB`, que representa el esquema de recursión primitiva *recr* sobre `AB`.
- `foldAB`, que representa el esquema de recursión estructural *foldr* sobre `AB`. Notar que puede definirse en términos de `recAB`.

Ejercicio 2

Definir la función `mapAB` que recibe una función f de un tipo `a` en un tipo `b`, y devuelve una función de árboles binarios de `a` en árboles binarios de `b`, la cual aplica la función f a cada uno de los nodos del árbol recibido.

Ejercicio 3

Definir la función `nilOCumple`, que dados una función comparadora, un elemento y un árbol, devuelve `True` si, o bien el árbol es `Nil`, o se satisface la comparación entre el elemento dado y la raíz del árbol.

Por ejemplo: `nilOCumple (<) 1 Nil` $\rightsquigarrow$ `True`
`nilOCumple (<) 1 (Bin Nil 2 Nil)` $\rightsquigarrow$ `True`
`nilOCumple (>=) 1 (Bin Nil 2 Nil)` $\rightsquigarrow$ `False`

Ejercicio 4

Definir las siguientes funciones sobre árboles binarios.

- `esABB`, que indica si un árbol binario es un árbol binario de búsqueda; es decir, si cumple que para cada nodo n, el valor de todos los elementos del subárbol izquierdo es menor o igual que el valor del nodo n y, a su vez, menores al valor de todos los elementos del subárbol derecho.
- `esHeap`, que dados un árbol binario con elementos de tipo `a` (es decir, un `AB a`) y una función f que compara valores tipo `a`, indica si un árbol binario es un heap; es decir, si cumple que para cada nodo n, la relación de n con todos sus descendientes cumple con la condición dada por f.

Ejercicio 5

Definir la función `completo`, que dado un árbol binario indica si es completo, es decir, si cumple con la siguiente fórmula, donde h indica la altura del árbol y n indica la cantidad de nodos.

$$2^h - 1 = n$$

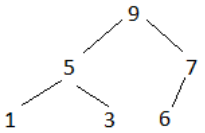
Ejercicio 6

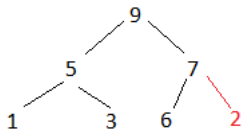
Definir las siguientes funciones sobre árboles binarios.

- `insertarABB`, que dados un árbol binario de búsqueda t con elementos de tipo `a` y un elemento e de ese mismo tipo `a`, devuelve un árbol binario de búsqueda que resulta de haber insertado el elemento e en el árbol t.
- `insertarHeap`, que dados un heap – representado por una función de comparación y un árbol binario con elementos de tipo `a` – y un elemento e de ese mismo tipo `a`, devuelve un heap representado por un árbol binario que resulta de haber insertado el elemento e en el heap.

  El elemento se debe insertar de manera tal que el árbol se mantenga balanceado. Para esto, los nodos se insertarán en el subárbol izquierdo hasta completarlo, y luego se pasará al derecho. De esta manera, la altura del árbol solo crecerá si se agrega un nodo a un árbol que ya estaba completo.

  Ej. Si se desea insertar el número 2 en el siguiente heap: el árbol resultante será:





  Notar que puede ser necesario realizar intercambios para mantener el invariante de heap.

  Sugerencia: definir y utilizar una función auxiliar para evitar repetir código.

Ejercicio 7

Definir la función `truncar`, que dado un árbol binario t y un número entero n, devuelve un árbol binario que resulta de haber truncado el árbol t a partir del nivel n. Es decir, el árbol resultante es igual a t hasta el nivel n, pero su altura es a lo sumo n (su altura será n a menos que t tenga menos de n niveles, en cuyo caso el árbol resultante será igual a t).

Tests

Parte de la evaluación de este Trabajo Práctico es la realización de tests. Tanto HUnit[1] como HSpec[2] 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.

[1] https://hackage.haskell.org/package/HUnit
[2] 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.