Departamento de
Informática e
Ingeniería de Sistemas
Universidad Zaragoza
1542

Práctica 6

Tecnología de Programación

Grado en Ingeniería Informática

Objetivos

- Comprender la definición y manejo de tipos y estructuras de datos en Haskell.
- Introducir los mecanismos de herencia y programación genérica.

Tarea

Diseña en implementa en Haskell las funciones que se piden a lo largo de la práctica.

1. Árboles binarios de búsqueda (BST)

En esta práctica, vas a programar en Haskell una serie de funciones que trabajan sobre árboles binarios de búsqueda (`https://en.wikipedia.org/wiki/Binary_tree`).

Deberás definir los tipos necesarios para implementar una estructura de datos que represente un árbol binario genérico (es decir, puede contener enteros, reales, cadenas de texto o incluso cualquier tipo de datos definido por el usuario).

Para construir un árbol de forma transparente, se deberán implementar las siguientes funciones:

- **empty** – Esta función devuelve un árbol vacío, sin ningún elemento.
- **leaf x** – Esta función devuelve un árbol que consta de una única hoja que contiene el elemento x.
- **tree x lc rc** – Esta función devuelve un árbol que contiene en la raíz el elemento x, con hijo izquierdo `lc` e hijo derecho `rc`. Tanto el hijo izquierdo como el hijo derecho son nodos del árbol, no elementos del tipo de datos genérico.
- **size t** – Esta función devuelve el número de elementos del árbol.

También deberás permitir que los árboles puedan ser visualizados por pantalla (con la alineación y formato que estimes oportuno) utilizando la función de Haskell `print`. Para ello deberás hacer que tu árbol binario instancie la clase Show, por lo que necesita implementar la función `show`.

Implementa toda la funcionalidad en un módulo independiente llamado **BinaryTree.hs**. Para probar tu módulo, puedes o bien utilizar el intérprete de Haskell `ghci` o bien hacer un programa principal. Prueba que el árbol es realmente genérico, es decir, prueba que el árbol funciona con diferentes tipos de datos.

2. Construcción de árboles binarios

Mediante las funciones que has definido puede construirse un árbol de forma manual, con código como el siguiente:

```
testTree = tree "R" (tree "HI" (leaf "NII") (leaf "NID"))
                    (tree "HD" (leaf "NDI") (leaf "NDD"))
```

que construye un árbol como éste:

```
"R"
 |- "HI"
     |- "NII"
     |- "NID"
 |- "HD"
     |- "NDI"
     |- "NDD"
```

Peso ese proceso es pesado y no garantiza que el árbol esté ordenado como requiere un árbol de búsqueda.

Implementa una función para añadir elementos a un árbol, de tal forma que se inserten ordenadamente (cada nodo siempre será mayor que todos los elementos del subárbol de la izquierda y menor que todos los elementos del subárbol de la derecha):

- **add t x** – Añade el elemento x al árbol t, devolviendo el árbol resultado.

A partir de ella, implementa una función que construya un árbol a partir de los elementos de una lista:

- **build xs** – Construye un árbol, comenzando con un árbol vacío e insertando sucesivamente los elementos de la lista xs.

Ahora puedes construir un árbol de la siguiente forma:

```
build [3, 2, 5, 1, 4]
3
|- 2
   |- 1
   |- <>
|- 5
   |- 4
   |- <>
```

Para poder implementar estas funciones, necesitarás asegurar que el tipo de dato de x (dado que los árboles son genéricos) tenga los operadores de comparación. Esto se consigue añadiendo una restricción para que x sea instancia de la clase **Ord** de Haskell.

3. Arboles binarios equilibrados

El proceso anterior genera un árbol que, aunque sea correcto, depende del orden en que se inserten los elementos:

```
build [3, 2, 5, 1, 4]

3
|- 2
   |- 1
   |- <>
|- 5
   |- 4
   |- <>

build [4, 3, 2, 5, 1]

4
|- 3
   |- 2
      |- 1
      |- <>
   |- <>
|- 5
```

El caso peor (patólogico) es utilizar una lista ordenada:

```
build [1..6]

1
|- <>
|- 2
   |- <>
   |- 3
      |- <>
      |- 4
         |- <>
         |- 5
            |- <>
            |- 6
```

Implementa una función que construya un árbol equilibrado a partir de los elementos de una lista:

- **buildBalanced xs** – Construye un árbol equilibrado, ordenando la lista y dividiéndola en dos por la mediana.

```
buildBalanced [1..6]

4
|- 2
   |- 1
   |- 3
|- 6
   |- 5
   |- <>
```

Para la implementación puedes buscar información de funciones de Haskell como **sort** o **splitAt**.

4. Recorrido de árboles binarios

Como ya sabes, hay tres tipos de recorridos que se pueden hacer sobre árboles binarios: en pre-orden, post-orden e in-orden.

Crea tres funciones (**preorder t**, **postorder t**, **inorder t**) que, dado un árbol **t**, devuelva una lista con todos los elementos del árbol en el orden correspondiente. Prueba que las tres funciones funcionan correctamente.

A partir de todas las funciones que ya tienes definidas, implementa una función que equilibre un árbol:

- **balance t** – Construye un árbol equilibrado a partir de otro cualquiera.

```
names  = build ["Adolfo","Diego","Juan","Pedro","Tomas"]

print names
"Adolfo"
|- <>
|- "Diego"
   |- <>
   |- "Juan"
      |- <>
      |- "Pedro"
         |- <>
         |- "Tomas"

print (balance names)
"Juan"
|- "Diego"
   |- "Adolfo"
   |- <>
|- "Tomas"
   |- "Pedro"
   |- <>
```

5. Búsquedas

El objetivo final de nuestra estructura de datos es realizar búsquedas. Para ello deberás implementar la siguiente función:

- **between t xmin xmax** – Busca en el árbol t y devuelve una lista con todos los elementos que están entre xmin y xmax (ambos inclusive). No debe recorrer todos los elementos del árbol, ni es necesario devolver la lista ordenada.

6. Pruebas

Te proporcionamos un fichero **BinaryTreeTest.hs** que prueba todas las funciones. Además de las pruebas que tú mismo hagas con el intérprete de Haskell, tus funciones deberán compilar con dicho archivo de pruebas. De lo contrario la calificación de la práctica será de 0.

Entrega

Como resultado de esta práctica deberás entregar **sólo el siguiente archivo**:

- **BinaryTree.hs** – que contenga la definición de la estructura de datos junto con todas las funciones arriba mencionadas.

Todas las funciones deben tener exactamente el mismo nombre expresado en este guión.

Todos los archivos de código fuente solicitados en este guión deberán ser comprimidos en un único archivo `ZIP` con el siguiente nombre:

- `practica6_<nip>.zip` (donde `<nip>` es el NIP de 6 dígitos del estudiante involucrado) si el trabajo ha sido realizado de forma individual.
- `practica6_<nip1>_<nip2>.zip` (donde `<nip1>` y `<nip2>` son los NIPs de 6 dígitos de los estudiantes involucrados) si el trabajo ha sido realizado en pareja. En este caso **sólo uno** de los dos estudiantes deberá hacer la entrega.

El archivo comprimido a entregar no debe contener ningún fichero aparte de los fuentes que te pedimos: ningún fichero ejecutable o de objeto, ni ningún otro fichero adicional. La entrega se hará en la tarea correspondiente a través de la plataforma Moodle.