Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Laboratorio de Lenguajes de Programación I (CI–3661)
Septiembre – Diciembre 2011

Proyecto III: Haskell

Se desea que usted implemente un algoritmo para compresión/descompresión de datos (sin pérdida de información) utilizando, para tal fin, un algoritmo de codificación de Huffman. Esta codificación intenta asignar a cada símbolo, de un conjunto de datos, un tamaño en bits que corresponda con la frecuencia con la que ocurre dicho símbolo. Siendo así, un símbolo con un número grande de ocurrencias tendrá un tamaño pequeño (posiblemente menor a su tamaño original). En cambio, un símbolo con un número pequeño de ocurrencias tendrá un tamaño grande (posiblemente mayor a su tamaño original). En promedio, la cantidad necesaria de bits para representar todos los datos suministrados se ve reducida, esto respecto a la misma si se representaran de forma tradicional (Por ejemplo, 8 bits fijos para caracteres ASCII).

Para la implementación de este algoritmo es necesario definir entonces un *Árbol de Huffman*. Este árbol guarda frecuencias acumuladas para símbolos en un conjunto de datos. Cada hoja, en este árbol, contiene un símbolo y la cantidad de ocurrencias del mismo en los datos suministrados (frecuencia). Luego, cada rama tendrá exactamente dos hijos y un valor de frecuencia acumulada. Dicha frecuencia será igual a la suma de las frecuencias de sus hijos.

La Figura 1. muestra un Árbol de Huffman para la siguiente entrada: “este es un ejemplo de un arbol de huffman”.

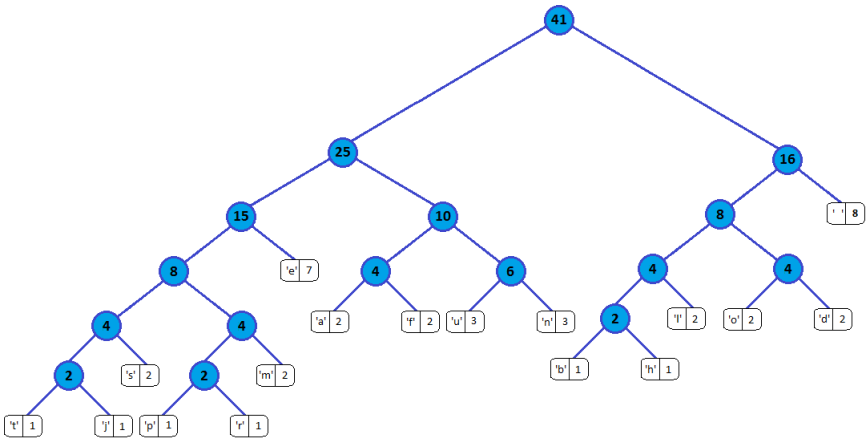

Figura 1: Árbol de Huffman para la entrada "este es un ejemplo de un arbol de huffman".

En Haskell definiríamos un Árbol de Huffman de la siguiente manera:

```
data Huffman a = Hoja Integer a | Rama Integer (Huffman a) (Huffman a)
```

Sobre estos Árboles de Huffman, deben definirse las siguientes funciones:

- `obtenerFrecuencia :: Huffman a -> Integer`

  Obtiene la frecuencia acumulada que se encuentra en la raíz del Árbol de Huffman suministrado.

- `crearRama :: Huffman a -> Huffman a -> Huffman a`

  A partir de dos Árboles de Huffman, crea un nuevo árbol que los tiene como hijos. La frecuencia acumulada del nuevo árbol es la suma de la de los nuevos hijos.

- `crearHuffman :: [(a, Integer)] -> Huffman a`

  Dado una lista que contiene un conjunto de símbolos y su cantidad de ocurrencias, construye un Árbol de Huffman que la represente.

- `generarAsociaciones :: Huffman a -> [(a,[Bool])]`

  Dado un Árbol de Huffman, devuelve una lista de tuplas con el conjunto de todos los símbolos ocurrentes en el árbol, junto a la codificación binaria de los mismos (La lista de booleanos representa la lista de bits de la codificación, donde `False` representa el cero y `True` el uno).

- `reconstruirHuffman :: [(a,[Bool])] -> Huffman a`

  Dada una lista de tuplas con un conjunto de símbolos, junto a una codificación binaria de los mismos (La lista de booleanos representa la lista de bits de la codificación, donde `False` representa el cero y `True` el uno). Construye un Árbol de Huffman tal que dichas asociaciones pudieran haber sido generadas. Para conservar las propiedades de un buen Árbol de Huffman, la frecuencia acumulada será asignada a cero (0) para todas las hojas y ramas del mismo (Nótese que es imposible extraer más información acerca de las frecuencias de los símbolos, sin embargo solo es importante para la futura codificación/decodificación la estructura del Árbol).

La construcción de Árboles de Huffman se hace de manera ambiciosa o *greedy*, a partir de un conjunto de Árboles de Huffman. Al principio se llena dicho conjunto con una hoja por cada símbolo que ocurra en los datos que se quieran analizar, junto a la frecuencia con la cual ocurren los mismos. Si el conjunto tiene un solo elemento, el mismo es el Árbol terminado. Si existe más de un elemento en el conjunto, se toman los dos Árboles de Huffman con la menor frecuencia acumulada y se crea una nueva rama con los árboles escogidos como hijos (recordando que la frecuencia acumulada de la rama creada será la suma de las frecuencias acumuladas de dichos hijos). A continuación se presenta un pseudo–código para dicha construcción

```
<Arbol de Huffman> construir ():

  1) C <- {Hoja elem frec | 'elem' es un elemento ocurrente en
                            los datos y 'frec' su frecuencia}

  2) Mientras C tenga más de un elemento:

    2.1) A, B <- Dos elementos con las mejores frecuencias en C.

    2.2) C <- C - {A, B} + {Rama con A y B como hijos}

  3) Retornar el elemento en C.
```

Nótese, que la escogencia de los Árboles con menores frecuencia (en el paso 2.1 del pseudo–código) debe realizarse utilizando una *Cola de Prioridades*. La prioridad sería tomada de menor a mayor por la frecuencia acumulada de los Árboles en el conjunto. La implementación de dicha cola queda libre, sin embargo es importante que incluya inserción y eliminación logarítmica con respecto al número de elementos en la cola, no lineal. Para tal fin se recomienda el uso de heaps en su implementación. Algunos heaps eficientes que pueden utilizarse son los *Binomial Heap*, *Leftist Heap*, *Fibonacci Heap*. Sin embargo, se puede

utilizar cualquier implementación diferente si se muestra que la inserción y la eliminación en el mismo es logarítmica con respecto al número de elementos en el heap.

Generar las asociaciones a partir de un Árbol de Huffman es seguir el camino necesario para llegar hasta el símbolo deseado, partiendo desde la raíz. Cada vez que el camino tome el hijo izquierdo, corresponderá a un cero (`False`). Cada vez que el camino tome el hijo derecho, corresponderá a un uno (`True`). Cuando se ha llegado a la hoja deseada se agrega un último uno (`True`) al camino. Tomando un ejemplo en la Figura 1., notemos que el caracter 'u' estaría representado por [`False`, `True`, `True`, `False`, `True`].

Finalmente, el objetivo es codificar y decodificar los datos, por lo que deben definirse dos funciones más:

- `codificar :: Eq a => [a] -> (Huffman a, [Bool])`

  Dado una lista de símbolos, devuelve una tupla. Esta tupla tiene como primer elemento el Árbol de Huffman generado a partir de la lista de símbolos dada. Como segundo elemento, tiene una lista donde cada uno de los símbolos en la entrada, ha sido reemplazado por su representación binaria y concatenados para formar una sola cadena. (Los elementos deben ser comparables por igualdad, para poder agruparlos y contar su cantidad de ocurrencias).

- `decodificar :: Huffman a -> [Bool] -> [a]`

  Dado un Árbol de Huffman y luego una lista que representa a una cadena de bits, devuelve una lista de símbolos que corresponde a la decodificación de dicha cadena con la información del árbol.

Se desea también que esta aplicación funcione para comprimir archivos. Para este motivo se les proveerá un módulo en Haskell `CompresorDeArchivos` que exporta las siguientes funciones:

- `leerDeArchivo :: String -> IO [Byte]`

  Dado el nombre de un archivo, lee sus contenidos y almacena el contenido en un arreglo de Bytes.

- `imprimirArchivo :: String -> [Byte] -> IO ()`

  Dado el nombre de un archivo y luego una lista de bytes, imprime dicha lista en el archivo con el nombre suministrado.

- `transformar :: Bool -> [Byte] -> [Byte]`

  Si el primer parámetro es `True`, transforma el segundo argumento a una nueva cadena de bytes representando el Árbol de Huffman generado y la cadena original codificada con el mismo. Si el primer parámetro es `False`, transforma el segundo argumento, interpretándolo como un Árbol de Huffman seguido de una cadena codificada. El resultado sería la cadena decodificada utilizando el Árbol leido.

El módulo incluye otras funciones que no son de exportación (solamente utilizadas en el módulo mismo). Así como la definición del tipo `Byte`.

La codificación de un Árbol de Huffman como una cadena de bytes, funciona de la siguiente manera: Sabemos, por la definición escogida, que toda cadena binaria asociada a un símbolo termina en `True`. Por lo tanto, el reverso de esa cadena siempre comienza en `True`. Este hecho será útil más adelante.

Para separar los bits que correspondan al camino en el Árbol de la representación binaria original del símbolo, se colocarán bits centinelas. Si al leer la cadena de bits aparece un `True`, el siguiente bit forma parte de un camino en el Árbol (reverso de la cadena binaria asociada a un símbolo). Si, de lo contrario, aparece un `False`, los siguientes 8 bits corresponden al símbolo el cual el camino anterior representa. Se toma por leido completamente el Árbol cuando, al comenzar a leer un camino en el Árbol, el primer bit es `False` (Esto se puede asegurar dado que, como se dijo anteriormente, toda camino leído comienza en `True`). Lo que sobre de la cadena de bits corresponde al mensaje original codificado.

Por ejemplo, tomemos nuevamente el símbolo `'u'`, de la Figura 1. Sabemos que el reverso de la cadena que lo representa es:

$[True, False, True, True, False]$.

Suponiendo que la representación binaria de `'u'` es el byte:

$[False, True, True, True, False, True, False, True]$ (su valor como ASCII),

entonces esta información será representada en la cadena binaria:

[True, $True$, True, $False$, True, $True$, True, $True$, True, $False$, False, $False$,

$True, True, True, False, True, False, True$]

Para la lectura y escritura en archivo se utiliza la librería `Data.ByteString`, que incorpora facilidades para leer los archivos y funciona mucho más velozmente que la librería estándar de Entrada/Salida.

Por último, debe crear un módulo Main, que exporte únicamente una función `main`, que reciba tres argumentos de la línea de comando (argv en C, args en Java). El primer argumento será el archivo fuente a leer. El segundo argumento será el archivo en el cual se ha de escribir. El tercer argumento será el *modo* de ejecución del programa y puede ser uno de entre `codificar` o `decodificar`.

Detalles de la Entrega:

La fecha límite para la entrega de este proyecto es para el Miércoles, 7 de Diciembre, a las 11:59pm. por Aula Virtual. La entrega debe consistir de sus archivos fuente de Haskell ('.hs') y una versión *imprimible* de los mismos ('.pdf'). Los mismos deben estar empaquetados en un archivo comprimido llamado `P3_G#.tar.gz`, (reemplazando el numeral (`#`) por el número de su equipo). Su código debe estar bien documentado utilizando la herramienta Haddock, explicando no solo las funciones y sus argumentos, sino los detalles de algoritmos que considere no triviales. Así mismo, su código debe estar bien modularizado. (Note que el módulo de `CompresorDeArchivos` necesita importar los módulos que tenga su implementación de Árboles de Huffman.) Además, su implementación debe utilizar funciones de orden superior siempre que sea posible.

Quedan prohibido el uso de librerías externas en su implementación, salvo por la librería `Data.List` y `System` (únicamente útil en el módulo `Main`). Su programa debe incluir un *Makefile* que permita compilar su programa a un ejecutable utilizando la instrucción `ghc`, asegurando que el mismo pueda ejecutarse con los argumentos que fueron especificados anteriormente. Cualquier código ajeno que incluyan en su implementación debe estar debidamente marcado y referenciado. Así mismo, todo el código utilizado debe ser comprendido a cabalidad, de manera que les sea posible explicar detalladamente su funcionamiento. **Recuerde que debe cumplir con la implementación de las funciones expuestas en el enunciado, respetando a cabalidad las firmas de las mismas.**