Procesamiento de archivos XML en Haskell

Lozano E. , Alvarado J & .Lasso H

19 de enero de 2014

1. Introducción

El propósito de nuestro proyecto es realizar un parseo de XML a una estructura manejable en Haskell. Este parseo se realiza para poder consultar y manipular datos totales, o ciertas especificaciones en una lista de 15438 Dispositivos con sus respectivas características. Esta lista es creada en base a la lectura de un archivo XML y a datos de tipos definidos por nosotros en Haskell, como lo son: Device, Group y Capability.

Haskell es un lenguaje de programación puramente funcional, éste permite decirle al computador no como tiene que hacer las tareas, sino más bien como están definidas. Es elegante y conciso ya que utiliza conceptos de alto nivel. Los programas Haskell son normalmente más cortos que los equivalentes imperativos por tanto es más eficiente su mantenimiento y además poseen menos errores.

2. Alcance

El alcance de nuestro proyecto Haskell es realizar la lectura de un archivo XML, manipular y analizar su contenido y representarlo en una estructura de tipo Lista en Haskell para realizar funciones de consulta o de busqueda.

3. Descripción

XML se puede considerar como un lenguaje que en sus documentos usa una estructura recursiva. Y eso a nuestro favor permite que Haskell manipule los módulos enteros del archivo con su mejor característica: Recursividad.

La recursividad en nuestro proyecto Haskell la usamos para poder manipular el archivo XML y reconocer que tipo de dato es el que encontramos en cada etiqueta de apertura y de cierre. Es decir las definidas con los simbolos de mayor o menor: <f>abc</f>. Estas etiquetas pueden ser en parejas (<f> ,</f>) o únicas (apertura y cierre) (<f ... />). Y que recursivamente pueden contener más parejas de etiquetas o texto plano.

Haskell incluye el soporte para los tipos de datos y funciones recursivas, Listas, Guardas y Tuplas. Por tanto el uso de este lenguaje es práctico a la hora de leer y representar un documento XML. Que es lo que buscamos.
De hecho, las combinaciones de estas características de Haskell pueden resultar en algunas funciones muy sencillas cuya versión usando lenguajes imperativos, puede llegar a resultar extremadamente tediosa de programar.

Gracias a esta funcionalidad de Haskell nuestra decisión de programación fue leer el documento XML y su conjunto de datos, para representarlo como una Lista. Pero no una Lista de Enteros o de Cadenas de Texto sino una Lista de un Tipo de Dato.
Haskell permite que definamos un Dato con sus características y a su vez el Tipo de Dato de las mismas. Por lo que definimos tres de estas en nuestro proyecto: Device, Group y Capability. Y al final lo que tuvimos como resultado fue una Lista que almacena Device.

```
ghci> :load "projectParserXMLHaskell.hs"
[1 of 1] Compiling Main             ( projectParserXMLH
askell.hs, interpreted )
Ok, modules loaded: Main.
ghci> xml <- readFile "cm1.xml"
ghci> let listaString = palabra xml
Loading package split-0.2.2 ... linking ... done.
ghci> let listaDevices = crearListaDevices listaString
ghci> :t listaDevices
listaDevices :: [Device]
ghci> length listaDevices
15438
```

Un Device tiene como características: idD de Tipo String, user_agent de Tipo String, fall_back de Tipo String y una Lista que almacena los grupos que contiene el Device de Tipo [Group].
Un Group tiene como características: idG de Tipo String y una Lista que almacena las capabilities que pertenecen a ese Group que son de Tipo [Capability].
Una Capability tiene como características: name de Tipo String y value de Tipo String.

Es de notar que es una forma muy eficiente de interpretar el XML, ya que como resultado obtenemos una Lista de Device que es más útil a la hora de manipular que el XML cómo tal. Por ejemplo un Device corto de la lista sería el siguiente:

```
ghci> listaDevices!!4
Device {idD = "sonyericsson_403_generic", user_agent = "DO_NOT_MATCH_SONYERICSSON_XHTML_BROWSER_4_0_3",
 fall_back = "sonyericsson_402_generic", groups = [Group {idG = "xhtml_ui", capabilities = [Capability
{name = "xhtml_file_upload", value = "supported"}]},Group {idG = "display", capabilities = [Capability
{name = "max_image_width", value = "120"},Capability {name = "resolution_width", value = "128"}]}]}
```

4. ¿Qué se implementó y qué no se implementó?

4.1. Lo implementado

Una vez leido y manipulado el contenido del archivo xml, la representamos en una lista de Devices que contiene elementos de tipo Device , cada Device contiene una lista de Groups que tienen elementos de tipo Group y cada Group contiene una lista de Capabilities que tienen elementos de tipo Capability. Con la lista de Devices ya obtenida podemos consultar la información de cada Device, Group o Capability. Un ejemplo sería el siguiente:

```
ghci> let consulta= [x|x <- listaDevices, buscarStringDevice x "can_skip_aligned_l
ink_row" == "false", buscarStringDevice x "brand_name" == "Orange"]
ghci> consulta
[]
ghci> let consulta= [x|x <- listaDevices, buscarStringDevice x "can_skip_aligned_l
ink_row" == "true", buscarStringDevice x "brand_name" == "Orange"]
ghci> consulta
[Device {idD = "orange_tr1_ver1", user_agent = "TR1/BSI AU.Browser/2.0 QO3C1 MMP/1
.0", fall_back = "generic_xhtml", groups = [Group {idG = "product_info", capabilit
ies = [Capability {name = "mobile_browser", value = "Teleca-Obigo"},Capability {na
me = "mobile_browser_version", value = "3.0"},Capability {name = "uaprof", value =
 "http://211.42.201.70/ua_profile/TR1.xml"},Capability {name = "model_name", value
 = "TR1"},Capability {name = "can_skip_aligned_link_row", value = "true"},Capabili
ty {name = "brand_name", value = "Orange"}]},Group {idG = "display", capabilities
= [Capability {name = "max_image_width", value = "168"},Capability {name = "resolu
tion_width", value = "176"},Capability {name = "resolution_height", value = "220"}
,Capability {name = "max_image_height", value = "200"}]},Group {idG = "bearer", ca
pabilities = [Capability {name = "max_data_rate", value = "9"}]},Group {idG = "str
eaming", capabilities = [Capability {name = "streaming_real_media", value = "none"
}]}]}]
ghci>
```

En estos dos ejemplos vemos a la función de busqueda retornando una lista vacia como una lista con un solo dispositivo que cumpla las características que buscamos. La función “buscarStringDevice” es para poder encontrar todos los Devices que tengan esas Capabilities.

4.2. Lo no implementado

Todos los requerimientos del proyecto han sido implementados.

5. Observaciones

- Aprender a programar en Haskell no fue sencillo, ya que para aquellos que estamos acostumbrados a lenguajes de programación imperativos es bastante abstracto aplicar en todas las funciones recursividad. Nos quedó claro que Haskell no posee estructuras de control y de flujo, y el porqué. Fue interesante leer e investigar sobre Haskell ya que salimos de la rutina de programación para entender que las estructuras de repetición (for, while...) no siempre son mejores que la recursividad.

6. Conclusiones

- Conocer y trabajar con nuevos lenguajes nos ayuda a comprender que distintas tareas se pueden resolver de distintas maneras y con diferentes criterios. Pero que siempre el fin es buscar simplificar la solución y sin complicaciones que muchas veces son evitables.
- Representar la información del archivo XML que nos proporciona información de distintos Dispositivos en una estructura sencilla, nos ayuda hacer consultas de una manera mas eficiente.
- La recursividad en muchos casos ahorra lineas de código que las estructuras de repetición, por lo que facilita el mantenimiento del código y a su vez permite disminuie la cantidad de errores.