CI4251 - Programación Funcional Avanzada
Tarea 3

Ernesto Hernández-Novich
86-17791
<emhn@usb.ve>

Mayo 26, 2016

1. *Buffer* de un editor

Durante la programación de un editor de texto es necesario modelar el *buffer* que contiene una línea particular mientras es modificada por el usuario. Las modificaciones se realizan según la posición del cursor en la línea.
Un modelo muy eficiente para el *buffer* de un editor sería

```
type Buffer = (String,String)
```

donde el primer `String` contiene los caracteres antes del cursor pero en orden **inverso**, y el segundo `String` contiene los caracteres después del cursor en el orden normal, e.g. suponiendo que el cursor sobre una letra es modelado con $\underline{X}$, la línea

Esto es un ejemplo, $\underline{d}$e como opera el buffer

se modelaría como

```
ejemplo = (" ,olpmeje nu se otsE","de como opera el buffer")
```

Defina las operaciones

```
empty      :: Buffer                    -- Buffer nuevo
cursor     :: Buffer -> Maybe Char      -- Leer bajo el cursor
insert     :: Char -> Buffer -> Buffer  -- ...antes del cursor
delete     :: Buffer -> Buffer          -- ...anterior al cursor
remove     :: Buffer -> Buffer          -- ...bajo al cursor
left       :: Buffer -> Buffer          -- Cursor a la izquierda
right      :: Buffer -> Buffer          -- Cursor a la derecha
atLeft     :: Buffer -> Bool            -- Extremo izquierdo?
atRight    :: Buffer -> Bool            -- Extremo derecho?
```

Implante estas operaciones y ofrezca suficientes propiedades QuickCheck para comprobar la correctitud de la implantación. Por supuesto que «suficientes» es un eufemismo por «use técnicas de análisis de cobertura para garantizar que sus casos de prueba ejercitan **toda** la librería».

Note que *ninguna* de las operaciones produce errores; esto quiere decir que, por ejemplo, `left` sobre un *buffer* vacío debe producir el mismo *buffer* vacío, así como `delete` al principio de la línea, no borra nada.

Finalmente, queremos mantener la implantación eficiente así que debe *evitar* el uso de la concatenación de listas (`++`).

2. Uso de `Parsec`

Un archivo "Literate Haskell" (`.lhs`) incluye código Haskell combinado con texto arbitrario. A efectos de este ejercicio supondremos que se trata de texto simple pero se desea convertirlo a HTML para su publicación en una página Web. Más aún, se adoptan las siguientes convenciones:

- Si una línea comienza con `*` se trata de un encabezado principal.
- Si una línea comienza con `#` se trata de un encabezado secundario.
- Un párrafo termina cuando haya una línea en blanco.
- Si una línea comienza *exactamente* con `>` seguido de espacio en blanco, se asume que el resto corresponde a texto del programa. Los dos caracteres al principio **no** deben conservarse.
- Los espacios en blanco son irrelevantes salvo el caso anterior. En otras palabras, los espacios en blanco entre palabras, antes del `*` al comienzo de una línea, etc. han de convertirse en un espacio en blanco sencillo.

Escriba un programa basado en un reconocedor `Parsec` tal que pueda ser utilizado para convertir los archivos `.lhs` con el formato antes descrito hacia HTML válido, tomando en cuenta que:

- Los encabezados principales y secundarios deben envolverse entre las marcas HTML `<h1>` y `<h2>`.
- Los párrafos sueltos deben envolverse entre las marcas `<p>`.
- Los segmentos continuos (múltiples líneas seguidas) con código Haskell deben envolverse entre las marcas HTML `<code>`.
- Los símbolos `<`, `>` y `&` tienen significado especial en HTML, por lo que deben ser convertidos a la entidad correspondiente.
- El resto del texto debe ser transportado "tal cual".

Su programa debe recibir uno o más nombres de archivo `.lhs` desde la línea de comandos y producir sendos archivos `.html` con los resultados de la transformación.