Laboratorio de Lenguajes de Programación

USB / CI-3661 / Sep-Dic 2016

(Programación Funcional – 35 puntos)

Funciones de Orden Superior (4 puntos)

Considere la función `unzip` discutida en clase

```
unzip :: [(a,b)] -> ([a],[b])
```

Se desea que Ud. implemente `unzip` de cuatro maneras diferentes:

1. Usando recursión directa

   ```
   unzipR = undefined
   ```

2. Usando un `foldr`

   ```
   unzipF = undefined
   ```

3. Usando `map`

   ```
   unzipM = undefined
   ```

4. Usando listas por comprensión

   ```
   unzipL = undefined
   ```

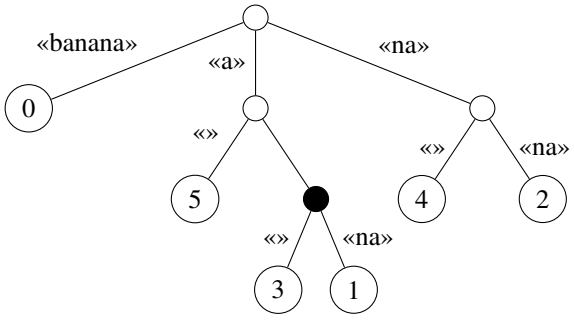
Árboles de Sufijos (17 puntos)

La búsqueda de cadenas («string matching») es un problema fundamental en la ciencia de la computación, aplicable desde el reconocimiento de lenguaje natural, pasando por la compresión de datos, hasta llegar al análisis de secuencias biológicas, sin excluir otras. El problema de *correspondencia exacta* quiere determinar si una cadena específica de búsqueda aparece en una cadena mucho más larga, y su ubicación.

Por ejemplo, la cadena «i miss mississippi» contiene seis ocurrencias de la cadena «s», en los índices 4, 5, 9, 10, 12 y 13; contiene tres ocurrencias de la cadena «is», en los índices 3, 8 y 11; dos ocurrencias de la cadena «issi», en los índices 8 y 11; y así el resto. Los índices comienzan por cero porque somos computistas, gente normal que cuenta bien.

Los *caracteres* componentes de una cadena *s* tienen índices *0..n-1* donde *n* es la longitud de *s*. Un **árbol de sufijos** permite representar una cadena arbitraria: cada hoja de un árbol de sufijos contiene un índice *entero* hacia *s*; cada subárbol de un nodo interno está etiquetado con una cadena. El árbol tiene la propiedad que la concatenación de las etiquetas en un camino desde la raíz hacia una hoja representa **exactamente un** sufijo de *s*, y el índice almacenado en la hoja es precisamente el índice del sufijo en *s*.

Un camino que visita la etiquetas s_1, s_2, …, s_k será denotado s_1 -> s_2 -> … -> s_k. Note que todos los sufijos son únicos porque tienen longitudes diferentes. Por ejemplo, para la cadena «banana», el árbol de sufijos sería:



Note que cada nodo interno representa una cadena *repetida*. Por ejemplo, el nodo marcado en negro corresponde al camino «a» -> «na», es decir la subcadena «ana». Esa subcadena aparece dos veces en «banana», específicamente en los *sufijos* «anana» (índice 1) y «ana» (índice 3). Eso explica las dos hojas que cuelgan del nodo, y las etiquetas asociadas a las aristas. Así mismo, note que el mismo camino nos indica que el substring «an» *también* aparece en los índices 1 y 3, pues «an» es prefijo de la concatenación de «a» y «na».

Buscar cadenas en un árbol de sufijos funciona buscando prefijos en los sufijos. Considere el árbol de sufijos para una cadena *t* y suponga que estamos buscando ocurrencias

de la cadena *s*. Se comienza por comparar la cadena *s* con las cadenas que etiquetan los subárboles de la raíz. Sea *a* una de esas cadenas; hemos de considerar tres casos:

1. Si *s* es prefijo de *a*, entonces es prefijo de *todo* sufijo indizado en las hojas de ese subárbol. Por ejemplo, si buscamos «n» y estamos revisando el subárbol «na», veremos que «n» es prefijo de «na» y de «nana». Así, la posición de «n» es, precisamente, las indicadas por las hojas (2 y 4).

2. Si *a* es prefijo de *s*, entonces la búsqueda debe continuar en el subárbol, pero primero debe removerse el prefijo *a* de *s*. Por ejemplo, si buscamos «ana» y estamos revisando el subárbol «a», veremos que «a» es prefijo de «ana». Así, tenemos que buscar recursivamente en el subárbol, pero usando «na», resultado de eliminar el prefijo «a» de «ana».

3. Si ninguno de los casos anteriores aplica, entonces la búsqueda falló en el subárbol, y hay que continuar con el *siguiente* subárbol, si quedara alguno.

Si ninguno de los subárboles satisface las condiciones antes descritas, entonces *s* no ocurre en *t*. Por ejemplo, si buscamos «nas»: no hay relación con «banana» (caso 3); no hay relación con «a» (caso 3); «na» es prefijo de «nas» (caso 2); «s» no se puede comparar con «» y no tiene relación con «na» (caso 3); como no quedan árboles, la búsqueda falla, i.e. «nas» no ocurre en «banana».

Tipo de Datos para Arboles de Sufijos

Ud. debe usar el siguiente tipo de datos para modelar árboles de sufijos

```
data SuffixTree = Leaf Int
                | Node [(String,SuffixTree)]
                deriving (Eq,Ord,Show)
```

Note que cada `Node` interno puede tener una cantidad arbitraria de subárboles, de allí el uso de una lista; cada subárbol está etiquetado con la cadena correspondiente. El índice almacenado en cada `Leaf` es no negativo.

Para nuestro árbol de ejemplo

```
t1 :: SuffixTree
t1 = Node [("banana", Leaf 0),
           ("a", Node [("na", Node [("na", Leaf 1),
                                    ("", Leaf 3)]),
                       ("", Leaf 5)]),
           ("na", Node [("na", Leaf 2),
                        ("", Leaf 4)])]
```

Ajá, ¿ahora qué hago?

Este ejercicio está dividido en cuatro partes, y diseñado para hacerlo en este orden. Además, las funciones solicitadas son tales que Ud. se vea obligado a escribirlas de una manera particular (hay muy pocas variantes) que le hagan ejercitar los conceptos de programación funcional.

Funciones de apoyo – faciliiiiiito (6 puntos)

1. (0.5 puntos) Escriba la función

   ```
   isPrefix :: String -> String -> Bool
   isPrefix p s = undefined
   ```

   que determina si `p` es prefijo de `s`.

2. (0.5 puntos) Escriba la función

   ```
   removePrefix :: String -> String -> String
   removePrefix p s = undefined
   ```

   que calcula el resultado de remover el prefijo `p` de `s`

3. (2.0 puntos) Escriba la función

   ```
   suffixes :: [a] -> [[a]]
   suffixes s = undefined
   ```

   que calcula todos los sufijos posibles de `s` en orden descendiente de longitud.

4. (1.5 puntos) Usando las funciones `isPrefix` y `suffixes`, escriba la función

   ```
   isSubstring :: String -> String -> Bool
   isSubstring s1 s2 = undefined
   ```

   que determine si `s1` es una subcadena de `s2`.

5. (1.5 puntos) Usando las funciones `isPrefix` y `suffixes`, escriba la función

   ```
   findSubstrings :: String -> String -> [Int]
   findSubstrings s1 s2 = undefined
   ```

   que produce los índices de *todas* las ocurrencias de `s1` en `s2` usando búsqueda directa inocente.

Buscando sobre árboles – nermel (4 puntos)

1. (1.5 puntos) Escriba la función

   ```
   getIndices :: SuffixTree -> [Int]
   getIndices t = undefined
   ```

   que produce los valores almacenados en las hojas del árbol. El orden de los valores no es importante – y no gana nada si los ordena.

2. (2.5 puntos) Aproveche la función `getIndices` para escribir la función

   ```
   findSubstrings' :: String -> SuffixTree -> [Int]
   findSubstrings' s t = undefined
   ```

   que implanta el método de búsqueda en árboles de sufijos descrito más arriba, para retornar los índices de *todas* las ocurrencias de `s` en `t`.

Construyendo árboles de sufijos – like a pro! (4 puntos)

1. (1.5 puntos) Escriba la función

   ```
   insert :: (String,Int) -> SuffixTree -> SuffixTree
   insert (s,i) t = undefined
   ```

   que produce el árbol de sufijos resultante de insertar el sufijo `s` con índice `i` en el árbol `t`

2. (2.5 puntos) Aproveche `insert` para escribir la función

   ```
   buildTree :: String -> SuffixTree
   buildTree s = undefined
   ```

   que construye el árbol de sufijos a partir de `s`. Es *obligatorio* expresar esta función usando el *fold* adecuado – soluciones con recursión directa o `map` no recibirán ningún punto aún si funcionan bien.

Folding trees – Do you feel lucky, punk? (3 puntos)

La subcadena más larga de una cadena que ocurre dos o más veces (posiblemente solapada), se denomina *subcadena repetida más larga*. Por ejemplo, la subcadena «ana» es la subcadena repetida más larga de «banana». Es cierto que «an» y «na» también aparecen dos veces, pero «ana» es más larga.

Defina la función

```
longestRepeatedSubstring :: SuffixTree -> [String]
longestRepeatedSubstring t = undefined
```

que encuentra la subcadena repetida más larga en el árbol `t`. Si hay más de una cadena repetida mas larga, debe encontrarlas todas.

«Hagenda» (14 puntos)

Picture Algebra (5 puntos)

Comenzaremos nuestra separación del problema, por modelar una representación de «imágenes» usando el tipo `Picture`.

```
type Height = Int
type Width  = Int

data Picture = Picture {
                 height :: Height,
                 width  :: Width,
                 pixels :: [[Char]]
             } deriving (Show)
```

La intención es poder contar con un tipo de datos simple, que nos permita componer «imágenes» complicadas a partir de «imágenes» sencillas. De aquí en más, aproveche la notación de registros, y las funciones auxiliares automáticas, para manipular el tipo de datos cuando tenga que usarlo en las funciones que va a escribir a continuación.

1. (0.25 punto) Nos interesa representar «imágenes» en base a caracteres, de modo que escriba la función

   ```
   pixel :: Char -> Picture
   pixel c = undefined
   ```

   tal que construya la «imagen» más simple posible.

2. (0.25 punto) Escriba la función

   ```
   above :: Picture -> Picture -> Picture
   above p0 p1 = undefined
   ```

   tal que si `p0` y `p1` son «imágenes» de la misma anchura, construya una nueva «imagen» en la que `p0` esté *encima* de `p1`. Si no tienen la misma anchura, debe fallar con el error `"can't 'above' different widths"`.

3. (0.25 punto) Escriba la función

```
beside :: Picture -> Picture -> Picture
beside p0 p1 = undefined
```

   tal que si `p0` y `p1` son «imágenes» de la misma altura, construya una nueva «imagen» en la que `p0` esté *antes* de `p1`. Si no tienen la misma altura, debe fallar con el error `"can't 'beside' different heights"`.

4. (0.25 punto) Escriba la función

```
toString :: Picture -> String
toString p = undefined
```

   tal que convierta `p` a una cadena simple susceptible de ser presentada en pantalla. **Debe** escribir esta función con el estilo *point free* (argumento implícito).

5. (0.25 punto) Aproveche la función `above` para escribir la función

```
stack :: [Picture] -> Picture
stack ps = undefined
```

   tal que si `ps` es una lista *no vacía* de «imágenes», se construya (de ser posible) una imagen combinada tal que la primera de la lista esté al tope. **Debe** escribir esta función con el estilo *point free* (argumento implícito).

6. (0.25 punto) Aproveche la función `beside` para escribir la función

```
spread :: [Picture] -> Picture
spread ps = undefined
```

   tal que si `ps` es una lista *no vacía* de «imágenes», se construya (de ser posible) una imagen combinada tal que la primera de la lista esté más a la izquierda. **Debe** escribir esta función con el estilo *point free* (argumento implícito).

7. (0.25 punto) Nos conviene tener la posibilidad de tomar un `String` y transformarlo en un `Picture` de altura 1 y longitud *n*, donde *n* es la longitud del `String`. Para ello, aproveche las funciones `spread` y `pixel`, escribiendo en estilo *point free* (argumento implícito) la función

```
row :: String -> Picture
row s = undefined
```

8. (0.25 punto) Escriba la función

```
blank :: (Height,Width) -> Picture
blank (h,w) = undefined
```

tal que se puedan construir «imágenes vacías» de anchura y altura arbitrarias. Una imagen estará vacía si está compuesta enteramente por espacios en blanco. **Debe** escribir esta función con el estilo *point free* (argumento implícito).

9. (1.0 punto) Es muy frecuente la necesidad de construir una imagen combinada vertical a partir de una lista de imágenes, intercalando imágenes en blanco entre ellas. Aproveche las funciones `above` y `blanks` para escribir la función

   ```
   stackWith :: Height -> [Picture] -> Picture
   stackWith h ps = undefined
   ```

   que construya la imagen combinada vertical a partir de `ps`, intercalando «imágenes» en blanco de altura `h`. **Debe** escribir esta función con el estilo *point free* (argumento implícito).

10. (1.0 punto) Es muy frecuente la necesidad de construir una imagen combinada horizontal a partir de una lista de imágenes, intercalando imágenes en blanco entre ellas. Aproveche las funciones `beside` y `blanks` para escribir la función

    ```
    spreadWith :: Height -> [Picture] -> Picture
    spreadWith w ps = undefined
    ```

    que construya la imagen combinada horizontal a partir de `ps`, intercalando «imágenes» en blanco de anchura `w`. **Debe** escribir esta función con el estilo *point free* (argumento implícito).

11. (0.5 punto) Dada una matriz de `Picture`, representada como una lista de listas, es necesario combinarlas (de ser posible) para construir una imagen combinada resultante. Esto es, cada lista dentro de la lista representa una *hilera* de imágenes que deben combinarse entre sí horizontalmente, para luego combinarse los resultados verticalmente y así producir la imagen resultante. Aproveche `stack` y `spread` para escribir la función

    ```
    tile :: [[Picture]] -> Picture
    tile pss = undefined
    ```

    que complete el «tiling». **Debe** escribir esta función con el estilo *point free* (argumento implícito).

12. (0.5 punto) Dada una matriz de `Picture`, representada como una lista de listas, es necesario combinarlas (de ser posible) para construir una imagen combinada resultante, intercalando imágenes en blanco entre ambas dimensiones. Esto es, cada lista dentro de la lista representa una *hilera* de imágenes que deben combinarse entre sí horizontalmente e intercalando imágenes en blanco de anchura `w`, para luego combinarse los resultados verticalmente intercalando imágenes en blanco de altura `h`, y así producir la imagen resultante. Aproveche `stackWith` y `spreadWith` para escribir la función

```
tileWith :: (Height, Width) -> [[Picture]] -> Picture
tileWith (h,w) pss = undefined
```

que complete el «tiling». **Debe** escribir esta función con el estilo *point free* (argumento implícito).

Años, meses, días y días de la semana (2.50 puntos)

Haskell cuenta con una batería interesante de librerías para manipular fechas y horas, sin embargo vamos a utilizar algunos tipos de datos más simplificados. Esto tiene el doble propósito de reducir la complejidad del problema, y explorar el sistema de tipos de Haskell.

Considere la siguiente representación para la información de un calendario:

```
type Day   = Int     -- Suponga que está entre 1 y 31
type Year  = Int     -- Suponga que es positivo
data Month = Enero
           | Febrero
           | Marzo
           | Abril
           | Mayo
           | Junio
           | Julio
           | Agosto
           | Septiembre
           | Octubre
           | Noviembre
           | Diciembre
           deriving (Show,Eq,Ord,Enum)

data DayName = Domingo
             | Lunes
             | Martes
             | Miercoles
             | Jueves
             | Viernes
             | Sabado
             deriving (Show,Eq,Ord,Enum)
```

Las operaciones con fechas se basan en algunas operaciones básicas que Ud. debe implantar en forma de funciones:

1. (0.25 puntos) La función

```
leap :: Year -> Bool
leap y = undefined
```

que indica si un año es bisiesto o no. Escriba la función como un *sólo* `if`.

2. (0.25 puntos) La función

```
mlengths :: Year -> [Day]
```

que para un año particular, devuelve una lista con la cantidad de días que tiene cada mes del año. Sólo puede escribir *una* lista literal como parte de la función.

3. (0.25 puntos) La función

```
jan1 :: Year -> DayName
jan1 y = undefined
```

que para un año particular indica cuál día de la semana correspondió al primero de enero. Por ejemplo,

```
ghci> jan1 2016
Viernes
```

El cálculo es muy simple: según el calendario Gregoriano, el primero de `Enero` del año uno fue `Lunes`. Entonces, todo lo que tiene que hacer es contar la cantidad de días desde el año 1 hasta el año en cuestión, teniendo cuidado con los años bisiestos. Puede escribir la función como una expresión aritmética cerrada, o aprovechando `leap` con recursión implícita – *no* puede usar recursión explícita.

4. (0.50) Aproveche `jan1` y `mlengths` para escribir la función de acumulación de días

```
mtotals :: Year -> [Int]
mtotals y = undefined
```

El primer día del año `y` corresponde a algún día particular de la semana. Si es `Lunes` diremos que el año comenzó con el día `1`, si es `Martes` diremos que el año comenzó con el día `2`, y así *módulo* 7. Esta función debe calcular la lista de días *acumulados* después que haya pasado cada mes. Por ejemplo, para el año 1969

```
ghci> mtotals 1969
[3,34,62,93,123,154,184,215,246,276,307,337,368]
```

porque ese año comenzó en `Miercoles` (por eso el `3`), y el resto de los números no son más que sumar los días de cada mes, i.e. 3, 3+31, 3+31+28, etc.

5. (0.50 puntos) Aproveche `mtotals` para escribir la función

```
fstdays :: Year -> [DayName]
fstdays y = undefined
```

que produzca la lista de días de la semana correspondientes al primer día de cada mes. Por ejemplo, para el año 1969

```
ghci> fstdays 1969
[Miercoles,Sabado,Sabado,Martes,Jueves,Domingo,
 Martes,Viernes,Lunes,Miercoles,Sabado,Lunes]
```

6. (0.25 puntos) Aproveche `fstdays` para escribir la función

```
fstday :: Month -> Year -> DayName
fstday m y = undefined
```

que produzca el día de la semana correspondiente al primer día del mes en un año particular. Por ejemplo,

```
ghci> fstday Agosto 1969
Viernes
```

7. (0.50 puntos) Aproveche `fstday` para escribir la función

```
day :: Day -> Month -> Year -> DayName
day d m y = undefined
```

que produzca el día de la semana correspondiente a una fecha en particular. Por ejemplo,

```
ghci> day 21 Abril 2003
Lunes
```

Picture Calendar (1.5 punto)

Nuestro objetivo es poder construir valores `Picture` que cuando sean presentados en pantalla se vean

```
        Octubre 2016
Do Lu Ma Mi Ju Vi Sa
                  1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

En efecto, vamos a imitar lo que hace el programa `cal` en cualquier sistema Unix. Si Ud. no usa Unix, debería, por su propio bien.

1. (0.25 puntos) Notará que bonito aparece el título del calendario alineado a la derecha, al igual que aquellos días del mes que tienen sólo un dígito. Es conveniente que escriba la función

   ```
   rjustify :: Int -> String -> String
   rjustify n s = undefined
   ```

   tal que a partir de `s` produce un nuevo `String` con suficientes espacios para que quede alineado a la derecha y ocupe exactamente `n` posiciones.

2. (0.50 puntos) Escriba la función

   ```
   dnames :: Picture
   dnames = undefined
   ```

   que construya el `Picture` correspondiente a los nombres de los días. Note que hay un espacio en blanco *antes* de cada etiqueta, inclusive antes del `Do`. Aproveche los combinadores de `Picture`, las instancias `Enum` y `Show` de `DayName`, y funciones de orden superior.

3. (0.25 puntos) Escriba la función

   ```
   banner :: Month -> Year -> Picture
   banner m y = undefined
   ```

   que construya el `Picture` correspondiente al membrete. Es necesario que esté alineado a la izquierda y que tenga el mismo ancho que `dnames` – más aún, si se me antojara cambiar `dnames`, `banner` debe seguir funcionando *sin* necesidad de hacerle cambios.

Si sus funciones están construidas correctamente, entonces

```
heading :: Month -> Year -> Picture
heading m y = banner m y `above` dnames
```

ha de funcionar para construir el encabezado del calendario con algo similar a

```
ghci> putStr $ toString $ heading Octubre 2016
         Octubre 2016
 Do Lu Ma Mi Ju Vi Sa
```

Un breve desvío (0.50 puntos)

Vamos a necesitar una función auxiliar para agrupar los elementos de una lista. Fíjese que para presentar el calendario de un mes, hay que agrupar los días «de siete en siete» – y cuando queramos presentar el calendario de un año (por supuesto que lo vamos a querer, porque `cal` lo hace), hay que agrupar los meses «de tres en tres».

Escriba la función polimórfica

```
group :: Int -> [a] -> [[a]]
group n xs = undefined
```

que preserva el orden de los elementos de `xs`, pero los organiza en grupos de a `n` elementos cada uno; el último grupo puede tener menos de `n` elementos. Por ejemplo

```
ghci> group 3 [1..10]
[[1,2,3],[4,5,6],[7,8,9],[10]]
```

Ahora viene lo dificil... (4.50 puntos)

Eventos en la agenda (1.25 puntos)

Considere el siguiente tipo de datos

```
data Evento = Evento {
                year        :: Year,
                month       :: Month,
                day         :: Day,
                nth         :: Int,
                description :: String
              }
            deriving (Show,Read)
```

que emplearemos para representar de manera precaria los eventos en una agenda. Los campos `description`, `year`, `month` y `day` tienen los significados evidentes, mientras que el atributo `nth` permite diferenciar varios eventos registrados para el mismo día, comenzando a contar desde uno.

La implantación de nuestra agenda requerirá un par de funciones auxiliares para interactuar con archivos, específicamente:

1. (0.50 puntos) Una función para leer desde un archivo

   ```
   loadEvents :: FilePath -> IO [Evento]
   loadEvents pathName = undefined
   ```

Puede suponer que el archivo a leer existe, es de texto, y puede contener *cero* o más líneas, cada una de las cuales tiene exactamente un `Evento`. La lista retornada como resultado de la lectura *debe* estar ordenada con los eventos desde el más antiguo hasta el más reciente, y de haber más de un evento ocurriendo en el mismo día, ordenados según el campo `nth`.

2. (0.50 puntos) Una función para escribir hacia un archivo una lista de eventos

   ```
   saveEvents :: FilePath -> [Evento] -> IO ()
   ```

   Si el archivo a escribir ya existiera, debe ser sobre escrito. Para cada `Evento` de la lista, se emitirá una línea de texto hacia el archivo. La lista *debe* escribirse en orden con los eventos desde el más antiguo hasta el más reciente, y de haber más de un evento ocurriendo en el mismo día, ordenados según el campo `nth`.

Aproveche las instancias `Show` y `Read` de los tipos de datos, las funciones convenientes en `System.IO`, así como la función `Data.List.sortBy`, para implantar estas dos funciones.

Finalmente, hará falta una función (0.25 puntos)

```
eventsOnMonth :: [Evento] -> Month -> [Day]
eventsOnMonth es m = undefined
```

que filtre la lista `es` extrayendo solamente los días del mes `m` en los cuales hay algún evento registrado. La lista resultante no puede tener elementos repetidos.

Presentando un calendario con eventos (1.50 puntos)

Nos interesa presentar el calendario de un mes particular, indicando aquellos días que tengan eventos registrados. Para eso es necesario implantar tres funciones:

1. (1.0 punto) Una función para construir la lista de «imágenes» correspondiente a *un* mes particular, marcando aquellos días que tengan al menos un evento registrado.

   ```
   pix :: DayName -> Day -> [Day] -> [Picture]
   pix d s ms =
   ```

   Esta función debe ser capaz de generar la lista de «imágenes» que representan un calendario como

   ```
                       1
    2  3  4  5  6  7  8
    9 10 11 12 13 14 15
   16 17 18 19 20 21 22
   23 24 25 26 27 28 29
   30 31
   ```

pero en forma de lista. Es decir, para cada día o espacio en blanco, debe generarse un `Picture`. Para ello, debe aprovechar los argumentos suministrados, considerando:

- El argumento `d` que corresponde al día en el cual comienza el mes, le permite determinar cuántas imágenes en blanco generar antes del primer día del mes.
- El argumento `s` corresponde al último día del mes, no sólo le permite determinar cuándo dejar de emitir imágenes para los días, sino comenzar a emitir imágenes en blanco.
- El argumento `ms` corresponde a aquellos días del mes en los cuales hay eventos registrados. Puede suponer que esta lista está ordenada de menor a mayor.
- La «imagen» correspondiente a un día debe ocupar exactamente tres espacios. Si el día tiene algún evento registrado, debe marcarlo con un >.

2. (0.25 puntos) Una función que construya la «imagen» que corresponde a la matriz de días del calendario

```
entries :: DayName -> Day -> [Day] -> Picture
entries dn d ds = undefined
```

   Aproveche las funciones para `group`, `tile` y `pix` para escribirla.

3. (0.25 puntos) Una función que (finalmente) construya la «imagen» con todo el calendario,

```
picture :: (Month,Year,DayName,Day,[Day]) -> Picture
picture (m,y,d,s,ms) = undefined
```

   Esta función se escribe, naturalmente, apoyándose en las funciones `entries` y `heading` de manera apropiada.

El Programa Principal (1.25 puntos)

Escriba las funciones necesarias para que el programa principal se comporte de la siguiente manera:

1. Al iniciarse el programa, debe leerse el archivo `hagenta.txt` en el directorio actual, cargando los eventos allí definidos. De lo contrario, comenzar con una lista vacía de eventos.
2. Determinar la fecha actual y presentar el calendario para el mes correspondiente, marcando aquellos días que tengan eventos registrados.

3. Presente un «prompt» de la forma

   ```
   2016 Octubre 15>
   ```

   en el cual puedan completarse las siguientes operaciones:

   - Si se presiona `'j'`, avanzar un *día*.
   - Si se presiona `'k'`, retroceder un *día*.
   - Si se presiona `'l'`, avanzar un *mes*.
   - Si se presiona `'h'`, retroceder un *mes*.
   - Si se presiona `'r'`, registrar un evento.
   - Si se presiona `'d'`, eliminar un evento.
   - Si se presiona `'q'`, salir del programa.
   - Si se presiona cualquier otra tecla, ignorarla y emitir un mensaje indicando las teclas válidas.

   En relación a esta funcionalidad:

   - Ante cualquier movimiento, es necesario presentar el calendario del mes apropiado y volver a presentar el «prompt» según el cambio.
   - Si el día a presentar tiene eventos registrados, estos deben ser listados inmediatamente *antes* del «prompt»
   - Para eliminar un evento, debe aparecer un «sub-prompt» que solicite el número de evento a eliminar.
   - Para registrar un evento, debe aparecer un «sub-prompt» que permita ingresar una línea con la descripción.
   - Al salir del programa, debe almacenarse la lista final de eventos en el archivo `hagenda.txt`
   - Para «limpiar» la pantalla, simplemente emita 24 líneas en blanco.

No le indicaremos la manera de estructurar las funciones de su programa principal, pero recuerde tomar en cuenta algunas de las técnicas indicadas en clase:

- Procure tener funciones auxiliares para los «prompt» y «sub-prompt», así como para las acciones asociadas a cada uno.
- Note que el «prompt» principal *no* debe esperar por un ENTER. Debe aceptar un caracter inmediatamente.
- Note que los «sub-prompt» *si* deben esperar por un ENTER, en un caso para tener un número y en otro caso para una cadena.
- Aproveche la recursión de cola y los argumentos como estado, para controlar el ciclo principal del programa. Más aún, si le parece conveniente, defina un tipo de datos a la medida para manejar el estado y así reducir la cantidad de argumentos que debe suministrar – si hace esto, asegúrese de aprovechar la notación de registros.

Detalles de la Entrega

La entrega se hará en un archivo `pH-<G>.tar.gz`, donde `<G>` debe ser sustituido por el número de grupo que le fue asignado. El archivo *debe* estar en formato TAR comprimido con GZIP – ignoraré, sin derecho a pataleo, cualquier otro formato que yo puedo descomprimir pero que *no quiero* recibir.

Ese archivo, al expandirlo, debe producir un *directorio* que *sólo* contenga:

- El archivo `unzip.hs` con la solución a la sección **Funciones de Orden Superior**
- El archivo `strees.hs` con la solución a la sección **Árboles de Sufijos**
- El archivo `hagenda.hs` con su implantación de la sección **Hagenda**. Debe escribirlo de manera tal que sea posible compilarlo haciendo

  ```
  ghc --make hagenda.hs
  ```

  para que produzca el ejecutable `hagenda`.

El proyecto debe ser entregado por correo electrónico a mi dirección de contacto a más tardar el domingo 2016-10-23 a las 23:59. Cada minuto de retraso en la entrega le restará un (1) punto de la calificación final.