Proyecto de programación funcional

Laboratorio de Lenguajes de Programación I

Enero–Marzo 2015

Índice general

---

1 Introducción

1.1 Mosaicos

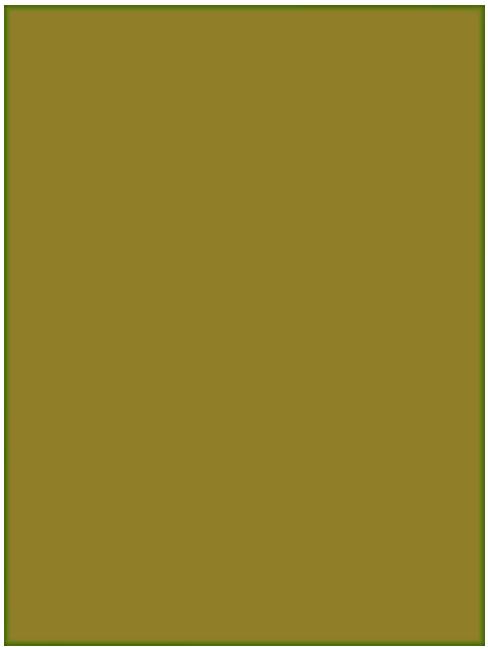
Un mosaico es una composición pictórica formada por una multitud de objetos pequeños que crean una imagen al juntarse en una cierta disposición geométrica. Se desea que elabore un

programa que facilite la elaboración interactiva de mosaicos digitales a partir de archivos de imagen. Los mosaicos digitales que serán utilizados en este proyecto serán rectangulares, y estarán compuestos únicamente de rectángulos alineados y adyacentes entre sí sin dejar espacios vacíos, y se elaborarán utilizando una imagen como punto de partida.

El procedimiento para elaborar mosaicos partirá de una imagen, y la dividirá por la mitad horizontal o verticalmente en dos partes, pudiendo dividirse a su vez las partes de la misma manera. Las partes finalmente resultantes serán dibujadas como un rectángulo con el color promedio de la región de la imagen original correspondiente a esa parte.

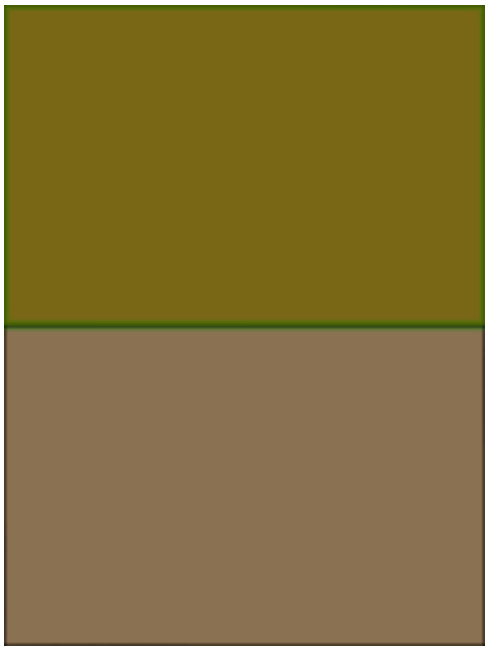
Por ejemplo, considere la siguiente foto de un emparedado de galletas con helado, Nutella™, y una cereza.

Inicialmente, a partir de la foto se puede construir un mosaico trivial: uno formado por un único rectángulo cuyo color es el color promedio de todos los píxeles de la foto[ˆresaltado]:



El borde verde que se observa en la imagen corresponde a que una parte del mosaico se considera *enfocada*, y se dibuja un trazo verde al rededor de la región enfocada. Además, cuando la parte del mosaico enfocada es un simple rectángulo, se indica visualmente tiñéndolo de amarillo. Como este mosaico trivial está compuesto únicamente de un rectángulo, se considera enfocado.

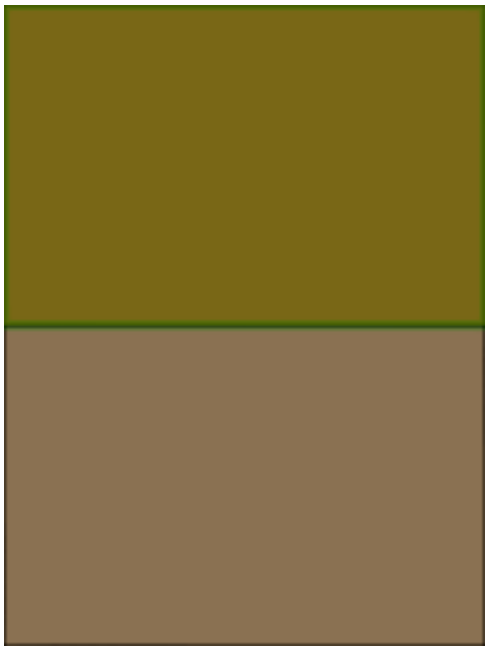
Luego, se puede dividir esa imagen por la mitad, por ejemplo horizontalmente, para obtener la siguiente imagen:



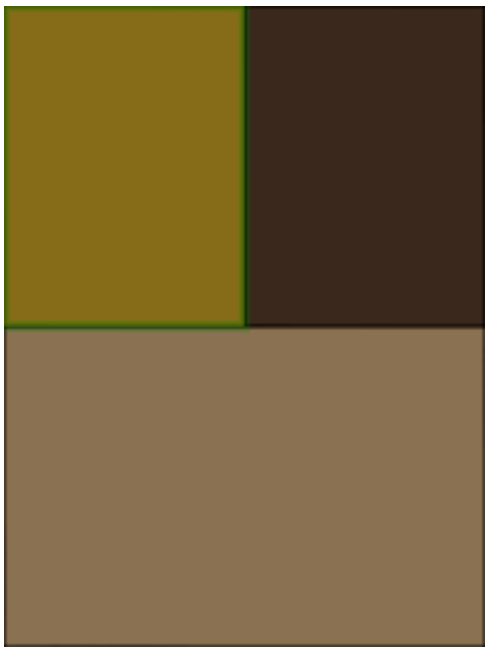
El mosaico está formado de dos rectángulos. En este caso, el foco permanece sobre la misma región, pero ha sido dividida en dos.

Cuando una región enfocada está dividida en dos, la primera sub-región se resalta en azul, y la segunda se resalta en rojo. En el caso de las divisiones horizontales, la parte encima de la división se considera la primera, y la parte debajo de la división es la segunda. En el caso de divisiones verticales, la parte a la izquierda de la división se considera la primera, y la parte a la derecha de la división se considera la segunda.

Desde ese punto, se puede trasladar el foco hacia la región de la imagen que se encuentra arriba de la división horizontal:

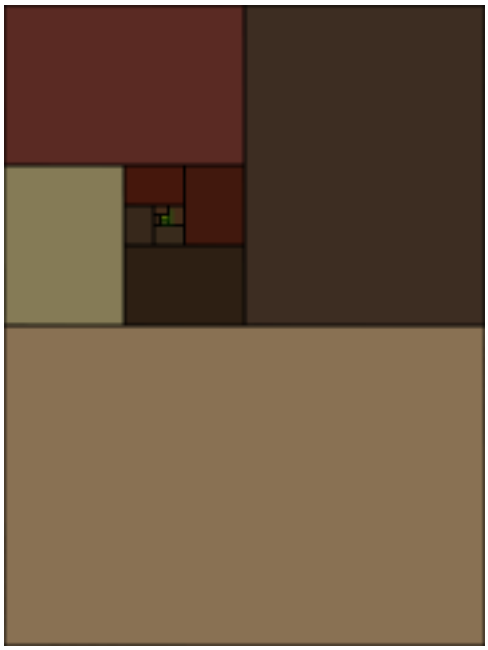


Puede observarse finalmente el color del rectángulo en la región inferior, cuyo color rosado pálido seguramente corresponde al promedio entre el blanco del mantel y el rosado del helado de fresa. Ahora, desde ese punto se puede realizar una división vertical y enfocar el lado izquierdo:

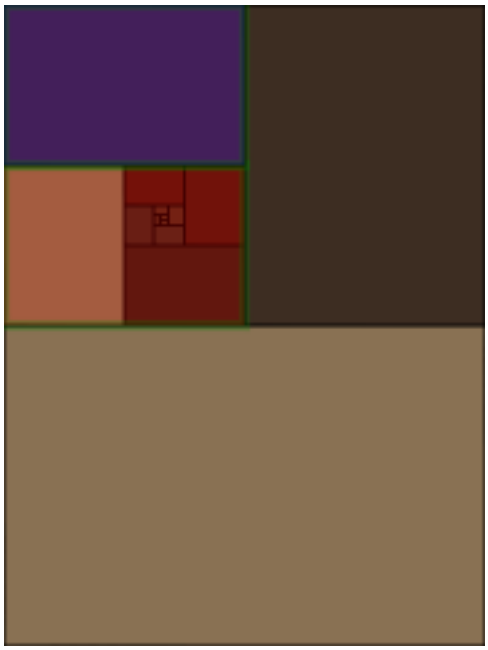


Note que el color marrón oscuro del rectángulo arriba y la derecha es el promedio del color del fondo oscuro de la foto en la esquina superior derecha, y de la galleta de chocolate. Luego se puede seguir con otra división horizontal y enfocar hacia abajo:

Note el color rojo del rectángulo en la esquina superior izquierda que viene del color de la cereza. Si se repite este proceso de subdivisión unas veces más, se puede llegar a algunas figuras interesantes:



Si el enfoque vuelve hacia afuera, las regiones enfocadas pueden tener muchas divisiones:



Si se efectúan muchas divisiones, se pueden obtener imágenes que se asemejan a la original:

1.2 Biblioteca `mosaico-lib`

Hackage es una colección pública de paquetes de *software* hecho con *Haskell* que incluye tanto bibliotecas como programas independientes. Se ha publicado en *Hackage* la biblioteca `mosaico-lib`, que consta de tres módulos con los siguientes propósitos:

- `Graphics.Mosaico.Diagrama`: Representación de distribuciones espaciales de rectángulos coloridos.
- `Graphics.Mosaico.Imagen`: Tipos y funciones para cargar archivos de imágenes.
- `Graphics.Mosaico.Ventana`: Creación y uso de ventanas interactivas para visualizar distribuciones de rectángulos.

Usted debe utilizar las definiciones exportadas de los módulos de este paquete para realizar este proyecto. **No** debe incluir el código de ese módulo en su proyecto, ni modificarlo de ninguna manera — de hecho, aunque el código de ese módulo está disponible, **no** se espera ni se recomienda que estudie su funcionamiento interno, y no es necesario conocer más que la documentación de esos módulos para poder utilizarlos en la elaboración de este proyecto.

Es esencial que estudie la documentación de ese paquete antes de comenzar a escribir su solución.

1.3 *Cabal* y `cabal`

Para facilitar la compilación de su proyecto, deberá elaborarlo en forma de un paquete de *Cabal*, el sistema de descripción de paquetes de software para *Haskell*. Se proveen junto con este enunciado los archivos que forman la estructura básica de un paquete de *Cabal* que puede usar como punto de partida para elaborar su proyecto. El archivo más importante de ese paquete es `mosaico-bin.cabal`, que es el archivo de descripción del paquete. En ese archivo debe sustituir `NOMBRES` y `EMAIL` por sus datos. Además, en la sección `other-modules` deben especificarse los nombres de todos los módulos que usted escriba como parte de su solución a este proyecto.

La sección `build-depends` del archivo de descripción del paquete especifica la lista de todos los paquetes de *Cabal* de los cuales su paquete depende. Toda biblioteca que su solución al proyecto utilice debe incluirse en esa lista. Puede utilizar cualquier biblioteca que desee; las únicas *necesarias* son `base` y `mosaico-lib`, que ya están en la lista de dependencias especificada en los archivos distribuidos junto con este enunciado. La herramienta de compilación se encargará de descargar, compilar e instalar todas las dependencias de su proyecto que sean paquetes de *Cabal*.

Antes que todo, al finalizar la instalación de GHC y `cabal-install`, debe ejecutar el comando

```
cabal update
```

para actualizar la lista de paquetes instalados en su sistema, y así hacer disponible las dependencias del proyecto.

Luego de instalar las dependencias externas de este proyecto, debe utilizar el comando

```
cabal install gtk2hs-buildtools
```

fuera del directorio del proyecto para instalar una herramienta de compilación necesaria para instalar las dependencias de *Haskell* de este proyecto.

Cuando un proyecto está descrito como paquete de *Cabal*, puede ser compilado utilizando el ejecutable `cabal`[ˆcabal], parte del sistema de automatización de compilación para paquetes de *Haskell*. Para usarlo, ubique un *shell* en el directorio donde esté el archivo de descripción de paquete de su proyecto (el archivo `mosaico-bin.cabal`), e invoque el comando

```
cabal sandbox init
```

Esto creará un entorno de compilación para su proyecto en ese directorio. Luego, invoque el comando


```
cabal install --only-dependencies
```

para instalar las dependencias del proyecto dentro del entorno de compilación recién creado,

```
cabal configure
```

para preparar a `cabal` para compilar su proyecto, y

```
cabal build
```

para compilarlo — al finalizar el comando, el texto de salida le indicará la ubicación del ejecutable resultante. Finalmente, si desea ejecutar una sesión interactiva de GHCi en el contexto de su proyecto desde la cual podrá usar todos los paquetes que haya declarado como dependencias, utilice el comando

```
cabal repl
```

Cada vez que modifique su código podrá repetir el comando

```
cabal build
```

para compilar con los últimos cambios hechos a la fuente. Para ejecutar el programa compilado, utilice el comando

```
./dist/build/mosaico/mosaico
```

desde el directorio principal del proyecto, donde `IMAGEN` es la ruta a un archivo de imagen. Puede usar imágenes en el formato PNG o JPEG.

Si necesita actualizar las dependencias del paquete, por ejemplo porque se publicara una versión actualizada del paquete `mosaico-lib`, debe repetir estos pasos en el directorio del proyecto:

```
cabal update
cabal install --only-dependencies --reinstall
cabal configure
cabal build
```

1.4 Dependencias externas

El paquete `mosaico-lib` requiere de la presencia de ciertas bibliotecas instaladas en su sistema que no están escritas en *Haskell*, por lo cual `cabal` es incapaz de instalarlas. En particular, debe instalar los paquetes para desarrollo con las bibliotecas *GTK*, *Cairo*, *Pango* y *GLib*. En sistemas basados en *Debian*, debería bastar el comando

```
sudo apt-get install libgtk2.0-dev libpango1.0-dev libglib2.0-dev libcairo2-dev
```

En el *wiki* de *Haskell* hay instrucciones para *Windows* y Mac OS X

1.5 Condiciones de entrega

Este proyecto debe ser realizado por cada alumno de CI3661 en grupos de a lo sumo dos integrantes. Debe entregar su solución en un archivo llamado `p1-XX-XXXXX_YY-YYYYY.tar.gz` (donde `XX-XXXXX` y `YY-YYYYY` deben ser sustituidos por los números de carné de los integrantes del grupo), o `p1-XX-XXXXX.tar.gz` para los casos excepcionales que realicen el proyecto en forma individual, enviado adjunto a un correo electrónico titulado *[CI3661] Proyecto 1* a las direcciones de *todos* los encargados del curso:

- Manuel Gómez manuel.gomez.ch@gmail.com
- David Lilue dvdalilue@gmail.com
- Ricardo Monascal rmonascal@gmail.com
- Wilmer Pereira wpereira@usb.ve

Su proyecto debe poder compilarse ejecutando estos comandos en el directorio resultante de descomprimir su proyecto:

```
cabal sandbox init
cabal install
```

En su solución, además del código que implementa el generador de mosaicos según esta especificación, incluya una *selfie* de su equipo en el archivo `selfie.png`. Corra el ejecutable resultante de su solución al proyecto con ese archivo, elabore manualmente un mosaico, y tome una captura de pantalla de la ventana con su mosaico. Guarde esta captura de pantalla en el archivo `mosaico.png` e inclúyalo también en su solución. Se recomienda que la resolución de estas imágenes no supere 600×600 píxeles.

Debe enviar su solución antes de la medianoche entre el martes 2015-05-19 y el miércoles 2015-05-20 en hora legal de Venezuela.

2 Especificación

Esta especificación se divide en tres secciones que corresponden a partes relativamente independientes del proyecto. Usted debe escribir un módulo de *Haskell* dentro de su paquete para cada una de esas secciones y agregarlo a la sección `other-modules` para que `cabal` sepa que debe tomarlo en cuenta durante la compilación del paquete. El código de cada módulo debe incluirse en un archivo separado dentro del directorio `src` y pueden incluirse entre sí en forma acíclica[1].

Más allá de las funciones que se especifican en este enunciado, usted puede escribir todas las definiciones auxiliares de tipos y funciones que considere convenientes para organizar su código.

2.1 Procesamiento de imágenes

Esta sección especifica el contenido del módulo `Imagen`.

Defina una función con la firma

```
subImagen
  :: Integer -> Integer
  -> Integer -> Integer
  -> Imagen -> Imagen
```

La expresión

```
subImagen xInicial yInicial anchura' altura' imagen
```

debe tener como valor resultante una `Imagen` cuya anchura y altura sean los valores de `anchura'` y `altura'`, respectivamente, y cuyos píxeles sean los de la región del tamaño adecuado tomada de la `imagen` pasada que comienza en la fila `yInicial` y en la columna `xInicial` contadas desde cero. `Imagen` es el tipo definido en el módulo `Graphics.Mosaico.Imagen` del paquete `mosaico-lib`.

---

[1] Es posible escribir varios módulos que se importen de manera cíclica, pero es engorroso e innecesario. Organice su código de forma tal que esto no sea necesario.

Utilizando la función `subImagen`, defina funciones con las siguientes firmas:

```
hSplit :: Imagen -> (Imagen, Imagen)
vSplit :: Imagen -> (Imagen, Imagen)
```

Las funciones `hSplit` y `vSplit` deben dividir por la mitad a la `Imagen` que reciben. `hSplit` debe producir una división horizontal: la imagen original se podría reconstruir ubicando el primer elemento de la tupla resultante **arriba** del segundo elemento de la tupla resultante. Análogamente, `vSplit` debe producir una división vertical: la imagen original se podría reconstruir ubicando el primer elemento de la tupla resultante a la **izquierda** de la tupla resultante.

Defina una función con la firma

```
colorPromedio :: Imagen -> Color
```

que, dada una imagen, calcule su color promedio. El color promedio es aquel `Color` cuyas componentes de color `rojo`, `verde` y `azul` sean los promedios de la componente de color rojo, verde y azul, **respectivamente**, de todos los píxeles de la imagen pasada como parámetro.

Por ejemplo, la siguiente expresión especifica una imagen de una sola fila con solamente tres píxeles:

```
Imagen
  { anchura = 4
  , altura  = 1
  , datos
      = [ [ Color { rojo =  14, verde = 35, azul = 250 }
          , Color { rojo =  75, verde = 25, azul =   0 }
          , Color { rojo = 120, verde =  0, azul = 250 }
          , Color { rojo =   0, verde =  3, azul = 100 }
          ]
        ]
  }
```

Calculando los promedios, se obtiene


```
rojoPromedio  = ( 14 + 75 + 120 +   0) / 4 = 209 / 4 = 52.25
verdePromedio = ( 35 + 25 +   0 +   3) / 4 =  63 / 4 = 15.75
azulPromedio  = (250 +  0 + 250 + 100) / 4 = 600 / 4 = 150
```

El color promedio de esa imagen es entonces

```
Color
  { rojo  =  52
  , verde =  16
  , azul  = 150
  }
```

Note que los resultados debieron ser redondeados luego de la división para poder usarse como componentes de color, que son enteros sin signo de 8 bits según la definición del tipo `Color`.

Ojo: Tenga mucho cuidado con las conversiones entre tipos numéricos al escribir esta función. Las operaciones aritméticas sobre el tipo `Word8` se efectúan *modulo* 256, por lo cual **no** debe realizar la suma de los valores de componentes de colores para el cálculo del promedio sin antes haber convertido esos valores a *otro* tipo numérico adecuado. En el ejemplo anterior se omitieron, por brevedad, las conversiones de tipos necesarias para obtener el resultado correcto, pero usted debe utilizarlas en su proyecto. **Verifique que los resultados de su implantación sean numéricamente correctos usando imágenes pequeñas y confirmando manualmente el resultado**.

2.2 Diagramas

Esta sección especifica el contenido del módulo `Diagramas`.

Defina una función con la firma

```
rectánguloImagen :: Imagen -> Rectángulo
```

que, dada una `Imagen`, produzca como resultado un `Rectángulo` cuya `imagen` sea la dada y cuyo `color` sea el color promedio de su `imagen`.

Considere el siguiente tipo enumerado usado para especificar si la orientación de una división es horizontal o vertical:

```
data Orientación
  = Horizontal
  | Vertical
  deriving Show
```

Utilizando las funciones `hSplit`, `vSplit` y `rectánguloImagen`, defina una función con la firma

```
dividir :: Orientación -> Rectángulo -> Maybe Diagrama
```

que, dada una especificación de orientación y un `Rectángulo`, retorne un cómputo en `Maybe` que produzca como resultado un `Diagrama` compuesto de una división del `Rectángulo` dado en dos `Rectángulos` nuevos. Las imágenes de los dos rectángulos nuevos deben ser el resultado de efectuar una división de la imagen del rectángulo dado con la orientación dada, y los colores de los dos rectángulos nuevos deben ser los colores promedio de sus respectivas imágenes. El cómputo retornado debe fallar (en el sentido de `Maybe`) si la imagen dada no tiene al menos dos píxeles de tamaño en la dimensión que se divide — esto es equivalente a decir que el cómputo retornado debe fallar si alguna de las imágenes producidas no tendría píxeles.

Defina una función con la firma

```
caminar :: [Paso] -> Diagrama -> Maybe Diagrama
```

que reciba una lista de `Pasos` y un `Diagrama`, y retorne un cómputo en `Maybe` que produzca como resultado el subárbol alcanzado, si lo hay, al recorrer el `Diagrama` dado desde la raíz siguiendo cada `Paso` en la lista dada, desde el primer elemento en adelante. Una lista de pasos vacía debe producir el mismo `Diagrama` dado. Si la lista de pasos dada no especifica un subárbol del `Diagrama` dado, el cómputo retornado debe fallar (en el sentido de `Maybe`).

Defina una función con la firma

```
sustituir :: Diagrama -> [Paso] -> Diagrama -> Diagrama
```

que reciba un `Diagrama` (el primero) a ser sustituido en una ubicación dada por una lista de `Pasos` dentro de otro `Diagrama` (el segundo), y retorne un `Diagrama` igual al segundo salvo porque el subárbol en la ubicación dada por la lista de `Pasos` ha sido sustituido por el primero.

La lista de `Pasos` especifica una posición en el árbol donde se hace la sustitución de igual manera a como se hace en la función `caminar`. Si se especifica una lista de `Pasos` que no corresponda con un subárbol existente en el árbol donde debe hacerse la sustitución, no se debe realizar ninguna sustitución. Es decir: si

```
Nothing = caminar ps d
```

entonces

```
d = sustituir d' ps d
```

En cambio, si

```
Just s = caminar ps d
```

entonces

```
Just d' = caminar ps (sustituir d' ps d)
```

Es decir: si se busca qué subárbol hay en una ubicación luego de sustituir el subárbol en esa ubicación por otro nuevo, se encuentra el nuevo — a menos que en esa ubicación no existiera, en cuyo caso no se encuentra nada.

Por ejemplo, si `r1`, `r2`, …, `r6` son `Rectángulos`, y dados estos `Diagramas`,

```
diagrama
  = (Hoja r1 :|: Hoja r2)
  :-:
    (Hoja r3 :|: Hoja r4)

diagrama'
  = Hoja r5
  :-:
    Hoja r6
```

entonces debe cumplirse que


```
sustituir diagrama' [Segundo, Primero] diagrama
  = (Hoja r1 :|: Hoja r2)
  :-:
    ((Hoja r5 :-: Hoja r6) :|: Hoja r4)
```

Además,

```
sustituir diagrama' [Primero, Segundo, Primero] diagrama
  = diagrama
```

ya que esa lista de pasos no lleva a la ubicación de ningún subárbol en `diagrama`.

2.3 Interacción

Esta sección especifica el contenido del módulo `Main`. Únicamente debe exportarse el símbolo `main`.

Defina una función con la firma

```
ciclo :: Ventana -> Diagrama -> [Paso] -> IO ()
```

que tome una `Ventana`, un `Diagrama` a dibujar en esa ventana, y una lista de `Pasos` que especifique cuál parte de ese `Diagrama` está siendo enfocada, y que retorne un programa que al ser ejecutado implante la interacción descrita en la introducción: debe intentar leerse una tecla de la ventana, y según la tecla que se haya pulsado, continuar mostrando un diagrama posiblemente distinto y posiblemente con otro enfoque. En particular:

- Si el foco se encuentra sobre un nodo del `Diagrama` correspondiente a una hoja y se recibe un evento de teclado correspondiente a una de las flechas (arriba, abajo, izquierda o derecha), se deberá dividir el rectángulo de la hoja enfocada, y continuar la ejecución con un diagrama en el cual la hoja enfocada ha sido sustituida por un nodo intermedio correspondiente a la división. La división será vertical si la tecla pulsada fue la flecha izquierda o la flecha derecha, y horizontal si no. El nuevo foco será sobre la hoja resultante de la división en la dirección pulsada: si fue la flecha izquierda o hacia arriba, será sobre el primer subárbol del nuevo nodo intermedio, y si fue a la derecha o hacia abajo, será sobre el segundo subárbol.

- Si el foco se encuentra sobre un nodo interno del `Diagrama` y se recibe un evento de teclado correspondiente a una de las flechas, se deberá mantener el `Diagrama` idéntico. Si el nodo intermedio enfocado representa una división horizontal y la tecla fue la flecha hacia arriba o abajo, el foco quedará sobre el primer o segundo subárbol, respectivamente, del nodo enfocado. El caso para divisiones verticales es análogo con las flechas a la izquierda y la derecha correspondiendo con el primer y segundo subárbol, respectivamente. Si se utiliza la flecha hacia la izquierda o derecha con el foco sobre una división horizontal, el foco permanecerá idéntico. El caso para divisiones verticales es análogo con las flechas de arriba y abajo: no alteran el foco.
- Si se pulsa la tecla *backspace* (borrar hacia atrás), se debe mantener el diagrama idéntico, y enfocar el nodo padre del nodo actualmente enfocado. Si el nodo enfocado era la raíz del diagrama, el foco no cambia.
- Si se pulsa la tecla `q`, se debe cerrar la ventana y terminar la ejecución.

Las divisiones y sustituciones deben hacerse usando las funciones que definió previamente.

Finalmente, defina el programa principal con la firma

```
main :: IO ()
```

que, al ejecutarse desde un *shell*, debe verificar que recibió exactamente un argumento de línea de comandos, usarlo como nombre de archivo para leer una imagen, crear una `Ventana`, y ejecutar el `ciclo` con la ventana creada, un `Diagrama` inicial que constará únicamente de un rectángulo hecho a partir de la imagen leída, y el foco en ese rectángulo.