Programación de Interfaces Gráficas de Usuario en Haskell con `AP.GUI`

Pablo López
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga

6 de mayo de 2011

1. La biblioteca `AP.GUI`

La biblioteca `AP.GUI` permite programar interfaces gráficas de usuario en Haskell. Un aplicación GUI es una aplicación interactiva que muestra componentes de una interfaz gráfica en la pantalla y ejecuta un bucle de eventos. Cada vez que se produce un evento, la aplicación lo despacha a la función adecuada (*callback*), que se asocia a un componente de la interfaz mediante un *listener*. El bucle de eventos termina cuando la aplicación ejecuta la acción `quit`.

El punto de entrada de una aplicación GUI es siempre la función `gui`:

```
gui :: IO () -> IO ()
```

que recibe un programa —o acción— como parámetro. La expresión `gui p`:

1. inicializa la biblioteca `AP.GUI`,
2. ejecuta el programa `p :: IO ()`, que posiblemente crea y muestra elementos de la interfaz e instala *callbacks* para los eventos que se desea manejar, y
3. atiende el bucle de eventos y los despacha a `p` hasta que `p` ejecuta la acción `quit :: IO ()`.

Si no se ejecuta `quit`, no se sale del bucle de eventos.

Aquí tenemos un ejemplo simple de aplicación GUI:

```
import AP.GUI
import System.Random -- para ejemplos posteriores

primerGUI :: IO ()
primerGUI = gui prog
  where
   prog = do
      putStrLn "Salida a consola, no al GUI..."
      quit
```

La anterior aplicación se ejecuta de forma inmediata. Mediante la acción `sleep`:

```
sleep :: Int -> IO ()
```

añadimos una espera de unos cuantos milisegundos para poder apreciar su ejecución. Mientras se ejecuta `sleep`, la aplicación no procesa los eventos, por lo que parece estar bloqueada.

```
espera :: Int -> IO ()
espera segundos = gui prog
 where
  prog = do
     putStrLn "Inicio del GUI..."
     putStrLn $ "esperando " ++ show segundos ++ " segundos..."
     sleep (1000 * segundos) -- pasar a milisegundos
     putStrLn "Fin del GUI."
     quit
```

2. Componentes, contenedores y controles

Los ejemplos anteriores dirigen su salida a la consola, no al GUI. Para dirigir la entrada y salida al GUI es necesario crear componentes (*widgets*) del GUI.

Un **componente** es un contenedor o un control. Un **contenedor** es una ventana (`Window`), un diálogo (`Dialog`), o un marco (`Frame`). Como su nombre indica, un contenedor contiene controles. Un **control** puede ser muchas cosas: `Label`, `Button`, `CheckButton`, `Edit`, `ScrollBar`, `Layout`, etc.

2.1. Atributos y propiedades

Cada componente puede tener una serie de propiedades (título, color de fondo, borde, texto, etc), que no son más que atributos de ese componente. Normalmente, las propiedades de un componente se establecen al crearlo (en el constructor del componente en cuestión); pero también se pueden consultar o, incluso, *modificar* más adelante.

Por ejemplo, una ventana (`Window`) tiene atributos para determinar su título (`title`) y el color de fondo (`bgColor`). El siguiente programa crea una ventana con un título y un color de fondo concretos:

```
ventanaDurmiente segundos = gui prog
  where
   prog = do
      window [ title =: "Durmiendo " ++ show segundos ++ " segundos..."
             , bgColor =: white
             ]
      update      -- para poder ver la ventana unos segundos
      sleep (1000 * segundos)
      quit
```

La acción `update :: IO ()` solicita que se refresque la visualización del GUI.

Podemos considerar un **atributo** como un «campo» que tiene un componente. Obviamente, no todos los componentes tienen los mismos atributos; por ejemplo, no todos los componentes tienen una barra de título. Cada atributo tiene un nombre con el que nos podemos referir a él (`title`, `bgColor`, etc); un tipo (`String`, `Color`, etc), que indica los valores que puede tomar, y tiene asociados un conjunto de componentes que lo poseen. Por ejemplo, el tipo de `title` es:

```
title :: Titled w => Attr w String
```

Este tipo nos indica que `title` es un atributo (`Attr`) que puede tomar una cadena (`String`) como valor y que se puede aplicar a un componente `w` siempre que éste sea instancia de la clase `Titled`. Los atributos están organizados mediante el sistema de clases de Haskell. Una clase define qué atributos tienen sus instancias. Con la orden:

```
Main> :i Titled
class (Widget w) => Titled w where title :: Attr w String
        -- Defined in AP.GUI.Lib.Attrs
instance Titled Dialog -- Defined in AP.GUI.Lib.Dialog
instance Titled Window -- Defined in AP.GUI.Lib.Window
```

Obtenemos una lista de los componentes que son instancia de la clase `Titled`; es decir, los componentes que tienen barra de título; en este caso, los contenedores ventana (`Window`) y diálogo (`Dialog`). También aparecen los atributos que tiene una instancia de la clase `Titled`; en este caso tenemos sólo el atributo `title`. Por su parte, el tipo del atributo `bgColor` es:

```
bgColor :: Background w => Attr w Color
```

lo que nos indica que este atributo toma como valor un color (el tipo `Color` también está definido en `AP.GUI`) y que se puede aplicar a los componentes que son instancia de `Background`.

Similarmente, para saber de qué atributos dispone el componente ventana (`Window`), podemos ejecutar la orden.

```
:i Window
```

La información que nos interesa está en las líneas de la forma

```
instance Clase Window
```

que nos indican qué atributos son aplicables a una ventana. Como podemos apreciar, la ventana tiene muchos atributos asociados, además de los de las clases `Titled` y `Background`.

Además de la información que podemos obtener con la orden `:i`, la biblioteca `AP.GUI` tiene documentación disponible en formato html. Esta documentación es más completa que la que podemos obtener con la orden `:i`, y tiene la ventaja de ser navegable. Toda la documentación html ha sido extraída directamente de los comentarios del código fuente mediante la aplicación `haddock`, que es el equivalente Haskell del `javadoc` de Java.

Ejercicio 1. Determina el tipo del atributo `position`, los valores que puede tomar y los componentes a los que se puede aplicar. Utiliza la orden `:i` y navega por la documentación html para encontrar toda la información necesaria.

Ejercicio 2. Determina qué atributos son aplicables a los diálogos (`Dialog`).

Sigamos analizando el programa anterior. La expresión:

```
window [title =: "ejemplo", bgColor =: white]
```

es el **constructor** del componente ventana. En este constructor indicamos valores iniciales para algunos de sus atributos. Para ello utilizamos el operador `=:`, cuyo tipo es:

```
(=:) :: Attr w a -> a -> Prop w
```

observa que `=:` toma un atributo y un valor para el mismo, y devuelve una **propiedad**. La diferencia entre atributo y propiedad es sutil: una propiedad es un atributo del que se conoce su valor, pero no a qué componente se aplica. Por ejemplo:

```
title =: "ejemplo" :: (Titled w) => Prop w
```

es una propiedad que se puede aplicar a un componente que tenga el atributo `title`; es decir, que sea instancia de la clase `Titled`.

Lo habitual es que los constructores reciban una lista de propiedades (atributos inicializados con `=:`) y devuelvan un componente con los atributos inicializados. Por ejemplo, el tipo del constructor `window` es:

```
window :: [Prop Window] -> IO Window
```

Observa que el componente en cuestión se devuelve como un valor impuro, dentro de la mónada `IO`. Esto es razonable por dos motivos:

- Por un lado, los atributos de un componente se pueden **modificar** durante la vida del mismo. Esto significa que los atributos de un componente conforman un **estado mutable** por efectos laterales, lo que conlleva al uso de una mónada.
- Por otro lado, los componentes del GUI se dedican a entrada/salida; se trata de una interacción gráfica dirigida por eventos, en lugar de un interacción textual dirigida por el flujo del programa. Es natural, por lo tanto, emplear la mónada `IO`.

Para acceder a los atributos de un componente utilizamos dos operadores: uno de lectura y otro de escritura. El operador de lectura es:

```
(?:) :: w -> Attr w a -> IO a
```

recibe como argumentos un componente `w` y un atributo, y devuelve el valor actual del atributo. El valor del atributo es mutable (impuro), por lo que se devuelve dentro de una mónada. Por ejemplo, la expresión:

```
ventana ?: title
```

devuelve como resultado un valor de tipo `IO String` con el título de la ventana.

El operador de escritura es:

```
(!:) :: w -> Prop w -> IO ()
```

recibe como argumentos un componente `w` y una propiedad; esto es, un atributo con su valor. Observa que no devuelve nada, pero tiene un efecto lateral: modifica el valor del atributo. Por ejemplo, la siguiente expresión modifica el título de una ventana:

```
ventana :! title =: "nuevo título"
```

El siguiente ejemplo muestra cómo podemos modificar el estado de una ventana durante la ejecución de un GUI:

```
mutarVentana segundos = gui prog
  where prog = do
        -- inicializar
           ventana <- window [title =: "dormida", bgColor=: blue]
           update
        -- leer
           titulo1 <- ventana ?: title
           putStrLn titulo1
           sleep (1000*segundos)
```

```
-- modificar
   ventana !: title =: "durmiendo"
   ventana !: bgColor =: red
   titulo2 <- ventana ?: title
   putStrLn titulo2
   sleep (1000*segundos)
   quit
```

El programa anterior modifica dos atributos de una ventana, usando para ello dos veces el operador `!:`. El operador `!!:`, cuyo tipo es:

```
(!!:) :: w -> [Prop w] -> IO ()
```

toma un componente y una lista de propiedades y aplica todas las modificaciones de la lista. Por ejemplo, en el programa anterior podríamos haber escrito:

```
ventana !!: [ title =: "durmiendo", bgColor =: red ]
```

Ejercicio 3. Estudia el atributo `size` y crea una ventana con un tamaño inicial. Duérmela un ratito, modifica su tamaño, y vuelve a dormirla. Observa que el tamaño de una ventana también se puede controlar con los atributos `height` y `width`: los tres atributos son métodos de la clase `Dimensions`.

Ejercicio 4. Utiliza el atributo `position` para crear una ventana que se vaya moviendo por la pantalla durante unos segundos. Puedes moverla en una trayectoria fija o aleatoria.

3. Manejo de eventos simples

Las anteriores ventanas simplemente se muestran, no podemos interactuar con ellas porque están «dormidas». Esto se debe a que la acción `quit` se ejecuta incondicionalmente, lo que nos obliga a utilizar acciones `sleep` para poder ver la ejecución. Una solución mejor consiste en ejecutar la acción `quit` condicionalmente, cuando se produzca un determinado evento. Esto permite eliminar las acciones `sleep`, de forma que las ventanas permanecen activas durante la ejecución de la aplicación. Por ejemplo, vamos a asociar un manejador de eventos para que la acción `quit` se ejecute al cerrar la ventana principal del GUI:

```
ventanaCerrable = gui prog
  where
   prog = do
      window [ title =: "Ventana cerrable"
             , on windowClose =: quit    -- manejador de eventos
             ]
      return ()
```

Podemos establecer un manejador de eventos en la lista de propiedades del constructor mediante la sintaxis:

```
on evento =: acción
```

Los eventos disponibles están agrupados en los siguientes *listeners*:

- eventos de teclado (`KeyListener`),

- eventos de ratón (`MouseListener`),
- eventos de ventana (`WindowListener`),
- eventos de acción o selección (`ActionListener`),
- eventos de foco (`FocusListener`) y
- evento de visualización (`PaintListener`).

Cada tipo de *listener* se encarga de cierto tipo de eventos. Los eventos de teclado y ratón llevan asociada cierta información extra (por ejemplo, el botón o tecla pulsados), por lo que la acción que los procesa recibe un argumento. Veremos cómo manejar eventos de teclado y ratón más adelante.

En este caso, nos interesa un evento de ventana (`WindowListener`). Consultando la documentación de `WindowListener`, obtenemos:

```
:i WindowListener
class (EventListener w) => WindowListener w where
  windowActivate :: Event w (IO ())
  windowDeactivate :: Event w (IO ())
  windowIconify :: Event w (IO ())
  windowDeiconify :: Event w (IO ())
  windowClose :: Event w (IO ())
  windowClosing :: Event w (IO ())
        -- Defined in AP.GUI.Lib.Event
instance WindowListener Dialog -- Defined in AP.GUI.Lib.Dialog
instance WindowListener Window -- Defined in AP.GUI.Lib.Window
```

El evento que nos interesa es `windowClose`, el que se produce al cerrar una ventana. El tipo de `windowClose` es:

```
windowClose :: WindowListener w => Event w (IO ())
```

Como los atributos, los eventos asocian a un componente `w` una acción que manejará el evento; en concreto, `windowClose` es un evento al que se puede asociar una acción de tipo `IO ()`. Este evento está disponible solo para los componentes `w` que sean instancia de la clase `WindowListener`: `Window` y `Dialog`. Por otra parte, la función `on`:

```
on :: EventListener =>Event w a -> Attr w a
```

es simplemente un envoltorio que permite ver el evento como un atributo asociado al componente. Este atributo se puede inicializar con el manejador, con lo que obtenemos una propiedad:

```
on windowClose =: quit :: WindowListener w => Prop w
```

En los siguientes ejercicios, podremos prescindir de esperas (`sleep`) y cerrar las aplicaciones manejando el evento `windowClose`.

Ejercicio 5. Crea una ventana de tamaño fijo (no redimensionable) y colocada en una posición concreta de la ventana.

Ejercicio 6. Cambia el color de fondo de una ventana según tenga o no el foco y según entre o salga el ratón de la misma.

Ejercicio 7. Define una función `ventanaAleatoria :: String ->IO Window` que devuelva una ventana aleatoria con un título dado. Por aleatoria se entiende que al menos los siguientes atributos deben estar inicializados aleatoriamente: dimensiones, posición y color de fondo.

Ejercicio 8. Escribe una función `collage :: Int ->IO ()` que cree `n` ventanas aleatorias y las muestre en la pantalla. Una de las ventanas aleatorias, elegida al azar, debe terminar la aplicación cuando se cierre (`quit`); el resto de ventanas se limitan a cerrarse ellas solas (`close`).

Ejercicio 9. Escribe un programa Haskell que muestre una ventana aleatoria en la pantalla. Cada vez que el usuario cierre la ventana, el programa decide aleatoriamente si se debe cerrar la aplicación, o si se debe cerrar la ventana y crear otra ventana aleatoria.

4. Añadiendo controles a un contenedor

En los ejemplos anteriores nuestras ventanas han permanecido vacías; hemos generado la salida directamente a la consola. Lo normal es gestionar casi toda la entrada/salida a través del GUI, mediante controles que aparecerán dentro de los contenedores.

El primer control que vamos a estudiar es la etiqueta (`Label`), que permite mostrar cierto texto en un contenedor. El constructor de las etiquetas es `label`:

```
label :: (Container w) => [Prop Label] -> w -> IO Label
```

La mayoría de los constructores de controles siguen el anterior esquema: toman una lista de propiedades para inicializar sus atributos y un contenedor `w` que es el padre del control. Como las etiquetas son instancia de la clase `Textual`, disponen del atributo `text`, que permite establecer el texto que debe mostrar la etiqueta:

```
holaMundo = gui prog
  where
   prog = do
      w <- window [ title =: "Saludo desde GUI"
                  , on windowClose =: quit
                  ]
      l <- label [text =: "¡Hola mundo!"] w
      return ()
```

El código anterior compila, pero no funciona correctamente (ni siquiera es suficiente añadir una acción `update`). ¿Cuál es el problema? Muy simple: hemos dicho que la ventana `w` contiene una etiqueta `l`, pero no hemos dicho dónde; no le hemos asignado posición dentro de la ventana. Para ello, hay que utilizar el atributo de la `layout`, que gobierna la disposición de los controles dentro de un contenedor:

```
holaMundo2 = gui prog
  where
   prog = do
      w <- window [ title =: "Saludo desde GUI"
                  , on windowClose =: quit
                  ]
      l <- label [text =: "¡Hola mundo!"] w
      w !: layout =: l -- posición por defecto
```

El anterior ejemplo coloca la etiqueta en una posición por defecto. Es posible controlar la disposición de un control mediante varias funciones. Por ejemplo, las funciones `east`, `west`, `north` y `south` permiten colocar la etiqueta en cualquiera de los puntos cardinales:

```
holaMundo3 = gui prog
  where
   prog = do
      w <- window [ title =: "Saludo desde GUI"
                  , on windowClose =: quit
                  ]
      l <- label [text =: "¡Hola mundo!"] w
      w !: layout =: east l  -- west, north, south
```

Para hacer más vistosa una etiqueta, podemos emplear los atributos `color` y `bgColor` para establecer el color del texto y del fondo. Además, mediante el atributo `font` de la clase `Typefaced`, podemos establecer el tipo de letra que queremos emplear, su tamaño y los efectos deseados (negrita, curvisa, subrayada). Por otro lado, las funciones que un control dentro de un contenedor se pueden combinar. En el siguiente ejemplo, colocamos la etiqueta en la esquina inferior izquierda.

```
holaMundo4 = gui prog
  where
     prog = do
        w <- window [ title =: "Saludo desde GUI"
                    , on windowClose =: quit
                    ]
        l <- label  [ text =: "¡Hola mundo!"
                    , font =: (bold . italic . arial)  26
                    , color =: yellow
                    , bgColor =: blue
                    ] w
        w!:layout =: west (attachBottom l)
```

Lo normal es que un contenedor contenga más de un control. Para especificar la disposición de los mismos, podemos utilizar operadores que combinan diferentes disposiciones, o funciones que reciben una lista de controles y los disponen en forma vertical (`vertical`), horizontal `horizontal` o cuadriculada `matrix`. El siguiente ejemplo emplea el operador binario `^.^` para colocar dos controles en un contenedor:

```
holaAdios = gui prog
  where
     prog = do
        w <- window [ title =: "Saludo desde GUI"
                    , bgColor =: white
                    , on windowClose =: quit
                    ]

        hola <- label [ text =: "¡Hola mundo!"
                      , font =: (bold . italic . arial) 26
                      , color =: yellow
                      , bgColor =: blue
```

```
                ] w

  adios <- label [ text =: "¡Adiós mundo!"
                 , font =: (bold . arial) 26
                 , color =: yellow
                 , bgColor =: red
                 ] w

  w!:layout =: hola ^.^ flexible (space (sz 10  10)) ^.^ adios
               -- ^ @ ^ : vertical
               -- < @ < : horizontal
               -- donde @ = . | - +
```

Los contenedores también pueden mostrar imágenes. Las imágenes se pueden cargar desde un fichero en formato `.gif`[1] con la función

```
loadImageFrom :: FilePath -> IO Image
```

Las imágenes no son controles, sino que se asocian a controles de la clase `Display` mediante el atributo `photo`. En el siguiente ejemplo colocamos dos imágenes asociadas a dos etiquetas:

```
tux :: IO ()
tux = gui prog
 where
   prog = do
      w <- window [ title =: "Imágenes de tux"
                  , on windowClose =: quit
                  ]

      imag1 <- loadImageFrom "super_tux.gif"
      lab1  <- label [ photo =: imag1
                     , bgColor =: cyan
                     ] w

      imag2 <- loadImageFrom "handy_tux.gif"
      lab2  <- label [ photo =: imag2
                     , bgColor =: cyan
                     ] w

      w !: layout =: flexible lab1 ^.^ flexible lab2
```

Ejercicio 10. El atributo `relief` de la clase `Bordered` permite indicar el tipo de borde que debe tener un control (`Raised`, `Sunkend`, `Solid`, `Ridge`, etc.). Modifica alguno de los ejemplos anteriores para asociar bordes a las etiquetas.

Ejercicio 11. Pinta el panel de una calculadora simple usando `matrix` como `layout`. Utiliza bordes para dar la apariencia de botones a las etiquetas.

[1] Si la imagen no está disponible en formato `.gif`, podemos utilizar un conversor.

5. Manejo de eventos de ratón y teclado

Los eventos que hemos manejado hasta ahora no llevan información asociada. Por el contrario, cuando se presiona o libera un botón del ratón, obtenemos información extra que indica qué botón se ha presionado o liberado y en qué posición del componente. Esta información se almacena en un dato de tipo `MouseEvent`, que es un registro con dos campos:

- el campo `mouseButton` indica qué botón lanzó el evento (`LeftButton`, `MiddleButton`, `RightButton`)
- el campo `mousePoint` representa mediante coordenadas planas (`pointX` y `pointY`) el punto del componente en que se encontraba el cursor del ratón.

El siguiente ejemplo genera un *log* en consola de las pulsaciones de los botones del ratón:

```
logPulsaciones = gui prog
  where
     prog = do
        w <- window [ title =: "Log de botones del ratón"
                    , on windowClose =: quit
                    ]
        w !: on mousePress =: procesaRaton

     procesaRaton :: MouseEvent -> IO ()
     procesaRaton r = putStrLn $ boton ++ "(" ++ x ++ ", " ++ y ++ ")"
         where
             boton = show . mouseButton $ r
             x = show . pointX . mousePoint $ r
             y = show . pointY . mousePoint $ r
```

Observa que al asociar el manejador del ratón hay que utilizar una acción que reciba un argumento (`MouseEvent ->IO ()`), y no una acción de tipo `IO ()` como en el evento de ventana `windowClose`.

Los eventos de ratón se pueden filtrar con los modificadores `left`, `right`, `middle`, `double` y `triple`, que permiten atender sólo ciertos eventos de ratón. El siguiente ejemplo diferencia entre pulsaciones simples y dobles del botón izquierdo:

```
logRatIzq = gui prog
  where
     prog = do
        w <- window [ title =: "Log de doble pulsación izquierda"
                    , on windowClose =: quit
                    ]
        w !: on (left mousePress) =: const procesaSimple
        w !: on (double (left mousePress)) =: const procesaDoble

     procesaSimple :: IO ()
     procesaSimple = putStrLn "¡Simple!"

     procesaDoble :: IO ()
     procesaDoble = putStrLn "¡Doble!"
```

En esta ocasión hemos utilizado la función `const` para deshacernos del argumento de tipo `MouseEvent`, dado que nuestros manejadores no lo utilizan.

Ejercicio 12. Usando los eventos de ratón, escribe una aplicación que tenga una barra de estado que informe en todo momento de la posición del ratón, si se encuentra dentro o fuera de la ventana, y del estado de sus botones, indicando si se ha producido una pulsación simple o doble. Observa que no todos los manejadores de eventos de ratón reciben un dato de tipo `MouseEvent` como argumento.

Cuando se produce un evento de teclado (`keyPress`, `keyRelease`) también es necesario recibir información sobre qué tecla se ha pulsado o liberado. Esta información se almacena en un registro `KeyEvent` con tres campos:

- el nombre de la tecla que lanza el evento (`keyText`),
- el carácter que tiene asociado la tecla (`keyChar`), que vale 0 si se trata de una tecla de control, y
- el código de la tecla (`keyCode`).

La siguiente función imprime la información del evento cada vez que se pulsa una tecla:

```
leeTecla = gui prog
  where
     prog = do
        w <- window [ title =: "Eventos de teclado"
                    , on windowClose =: quit
                    ]

        w !: on keyPress =: print
```

La función anterior no es capaz de detectar combinaciones de teclado, como por ejemplo `Alt+x`; las combinaciones se procesan secuencialmente: primero `Alt` y después `x`. Para trabajar con combinaciones de teclado es necesario utilizar los modificadores `alt`, `control` y `shift`.

```
leeAltTecla = gui prog
  where
     prog = do
        w <- window [ title =: "Eventos de teclado ALT+..."
                    , on windowClose =: quit
                    ]

        w !: on (alt keyPress) =: print
```

Ejercicio 13. Escribe una aplicación visual que represente el nombre, letra y código de las teclas que se pulsan. Te será útil para determinar el código asociado a ciertas teclas, que después podrás utilizar para asociar acciones a teclas concretas.

6. Controles interactivos y otros contenedores

Aunque es posible asociar eventos a etiquetas, lo más habitual es que la interacción en un GUI se produzca a través de otros controles como botones, botones de radio, menús, cajas de entrada, etc. La mayoría de estos controles se limitan a tener un constructor que indica los valores iniciales de sus atributos y qué contenedor es su padre. También es habitual que en un GUI sofisticado aparezca más de un contenedor; sobre todo cajas de diálogo y marcos.

6.1. Botones y diálogos

El constructor de botones:

```
button :: (Container w) => [Prop Button] -> w -> IO Button
```

permite inicializar un botón e indicar en qué contenedor debe aparecer. Los botones son instancia de la clase `ActionListener`, por lo que disponen del evento `action`, que se dispara al presionar el botón. Mediante el envoltorio `on`, podemos asociar una acción de tipo `IO ()` a tal evento; por ejemplo, el constructor:

```
q <- button [text =: "Abandonar", on action =: quit] w
```

crea un botón para cerrar el GUI.

Crear una caja de diálogo es muy simple; basta utilizar el constructor:

```
dialog :: (Container w) => [Prop Dialog] -> w -> IO Dialog
```

por ejemplo:

```
d <- dialog [title =: "Un diálogo"] w
```

Observa que una caja de diálogo siempre tiene un contenedor padre (`w`); mientras que las ventanas no tienen padre:

```
window :: [Prop Window] -> IO Window
```

de hecho, el contenedor ventana es el único que no tiene padre. Esto significa que nuestros GUIs siempre tendrán al menos una ventana como contenedor principal. Observa también que cualquier contenedor puede ser padre (no sólo las ventanas). Cuando un contenedor tiene un hijo, el foco de la aplicación pasa a éste, y no volverá al padre hasta que el hijo se haya cerrado.

El siguiente ejemplo permite crear un número indeterminado de cajas de diálogo encadenadas:

```
encadenaDialogos :: IO ()
encadenaDialogos = gui prog
   where
      prog = do
         w <- window [ title =: "Ventana padre"
                     , on windowClose =: quit
                     ]

         q <- button [ text =: "Abandonar"
                     , on action =: quit
                     ] w

         b <- button [ text =: "Nuevo Diálogo"
                     , on action =: creaDialogo w 1
                     ] w

         w!:layout =: b <.< q

creaDialogo :: Container w => w -> Int -> IO ()
creaDialogo w n = do
```

```
    d <- dialog [ title =: "Diálogo " ++ show n
                ] w
    b1 <- button [ text =: "Cerrar"
                 , on action =: close d
                 ] d
    b2 <- button [ text =: "Nuevo Diálogo"
                 , on action =: creaDialogo d (n+1)
                 ] d
    q  <- button [ text =: "Abandonar"
                 , on action =: quit
                 ] d

    d !: layout =: b1 <.< b2 <.< q
```

Observa que en el ejemplo anterior, el contenedor más reciente retiene el foco. Esto no ocurre cuando se crean varias ventanas: como las ventanas no tienen padre, todas pueden recibir el foco (excepto si lo tiene alguno de sus hijos).

6.2. Una ventana de login: cajas de entrada y botones de selección

Vamos a diseñar una ventana de *login* completa. La ventana pedirá al usuario su nombre de usuario, la clave (que podrá ocultarse mediante asteriscos) y tendrá tres botones que permitirán realizar el *login*[2], limpiar los campos de entrada y cancelar respectivamente. Nuestra ventana de *login* contendrá cajas de entrada y un botón de selección.

Puesto que se trata de una ventana relativamente pequeña y de contenido concreto, vamos a deshabilitar la posibilidad de cambiar su tamaño. Para ello utilizamos el atributo Booleano `resizable` (disponible en ventanas y diálogos).

Una caja de entrada permite introducir una línea de texto. La clase `Entry a` implementa las cajas de entrada; se trata de una clase polimórfica, pues la entrada puede tomar un valor de cualquier tipo `a` (lo más habitual es que el tipo sea `String`). El valor de la caja de entrada se almacena en un atributo `value`. El atributo Booleano `password`, que no pertenece a ninguna clase, permite ocultar el contenido de la caja de entrada en pantalla.

Los botones de selección (`Checkbutton`) son instancia de la clase `Checkable`, que define un atributo Booleano `checked` que indica si el botón está seleccionado o no.

```
login = gui prog
  where
    prog = do
       w <- window [ title =: "Inicio de sesión"
                   , resizable =: False
                   , on windowClose =: quit
                   ]

       ln <- label [ text =: "Nombre" ] w
       n  <- entry [ value =: "su nombre"
                   , font =: arial 14
                   ] w
```

[2]Lo que haremos será escribir en la consola el *login* deseado.

```
        lc <- label [ text =: "   Clave" ] w
        c  <- entry [ value =: "su clave"
                    , font =: arial 14
                    , password =: True
                    ] w

        ocultar <- checkButton [ text =: "ocultar"
                               , checked =: True
                               , on action =: (c !: password <: not)
                               ] w

        iniciar <- button [ text =: "Iniciar"
                          , on action =: do
                                 imprime n c
                                 quit
                          ] w

        limpiar <- button [text =: "Limpiar"
                          , on action =: limpia n c
                          ] w

        cancelar <- button [text =: "Cancelar"
                           , on action =: quit
                           ] w

        w!:layout =: west (ln <.< n)  ^.^
                     west (lc <.< c <.< ocultar) ^.^
                     iniciar <.< hSpace 20 <.< limpiar <.< hSpace 20 <.< cancelar


imprime :: Entry String -> Entry String -> IO ()
imprime nombre clave = do
           usr  <- nombre ?: value
           pass <- clave ?: value
           putStrLn $ "usuario: " ++ usr
           putStrLn $ "clave: "   ++ pass

limpia :: Entry String -> Entry String -> IO ()
limpia nombre clave = do
           nombre !: value =: ""
           clave  !: value =: ""
```

En la acción del botón de selección `ocultar` hemos utilizado el operador `<:`:

```
(<:) :: Attr w a -> (a -> a) -> Prop w
```

que permite actualizar un atributo aplicándole a su valor actual la función que aparece como segundo argumento (en nuestro caso, `not`).

7. Eligiendo la sesión: botones de radio y marcos

Vamos a ampliar nuestra ventana de *login* para que nos permita elegir el tipo de sesión que queremos iniciar. Para ello utilizaremos botones de radio (`RadioButton`). Los botones de radio, al igual que los botones de selección, son instancia de la clase `Checkable`, por lo que pueden estar seleccionados o no (atributo `checked`). Como otros controles, los botones de radio son instancia de la clase `Able`, que define el atributo Booleano `enabled`, para indicar si el control está o no habilitado.

Lo habitual es que la selección de los botones de radio sea excluyente. Para ello es necesario agrupar los botones de radio en un grupo mediante un `RadioGroup`, que tiene un atributo `radioButtons` cuyo valor es la lista de botones de radio que integran el grupo. Los `RadioGroup` son instancia de la clase `Selection`, que cuenta con el atributo entero `selection`, que devuelve la selección actual (numerada desde 0).

Es frecuente que los grupos de los botones de radio se representen enmarcados. Podemos utilizar un contenedor marco (`Frame`), al que asociaremos un borde adecuado. Además, los marcos cuentan con el atributo `layout`, que permite indicar qué elementos contiene el marco y dónde se colocan. Esta última característica es muy interesante, pues permite introducir varios marcos dentro de un contenedor y realizar una disposición jerárquica de los elementos. En nuestro ejemplo, hemos agrupado los controles en tres marcos, uno de ellos con borde invisible.

```
loginSesion = gui prog
  where
    prog = do
       w <- window [ title =: "Inicio de sesión"
                   , resizable =: False
                   , on windowClose =: quit
                   ]

   -- marco de identificación de usuario

       ln <- label [ text =: "Nombre" ] w
       n  <- entry [ value =: "su nombre"
                   , font =: arial 14
                   ] w

       lc <- label [ text =: "   Clave" ] w
       c  <- entry [ value =: "su clave"
                   , font =: arial 14
                   , password =: True
                   ] w

       ocultar <- checkButton [ text =: "ocultar"
                              , checked =: True
                              , on action =: (c !: password <: not)
                              ] w

       fusr <- frame [ relief =: Groove
                     , layout =: west (ln <.< n) ^.^
                                 west (lc <.< c <.< ocultar)
                     ] w
```

```
 -- marco de tipo de sesión

    gnome  <- radioButton [ text=: "Gnome" ] w
    kde    <- radioButton [ text=: "KDE", enabled =: False ] w
    xfce   <- radioButton [ text=: "Xfce" ] w
    xmonad <- radioButton [ text=: "Xmonad", checked =: True ] w

    let sesiones = [ gnome, kde, xfce, xmonad ]
    sesionesTxt <- mapM (?: text) sesiones

    grpSesiones <- radioGroup [ radioButtons =: sesiones ]

    fses <- frame [ relief =: Groove
                  , layout =: west gnome  ^.^
                              west kde    ^.^
                              west xfce   ^.^
                              west xmonad
                  ] w

-- marco de botonera

    iniciar <- button [ text =: "Iniciar"
                      , on action =: do
                           imprime n c
                           numSesion <- grpSesiones ?: selection
                           putStrLn ("sesion: " ++ sesionesTxt  !! numSesion)
                           quit
                      ] w

    limpiar <- button [text =: "Limpiar"
                      , on action =: do
                           limpia n c
                           xmonad !: checked =: True
                      ] w

    cancelar <- button [text =: "Cancelar"
                       , on action =: quit
                       ] w

    fbot <- frame [layout =: iniciar   ^.^
                             vSpace 15 ^.^
                             limpiar   ^.^
                             vSpace 15 ^.^
                             cancelar
                   ] w

-- disposición de los marcos
```

```
w!:layout =:   fusr ^.^
               fses <.< hSpace 40 <.< fbot
```

Un detalle interesante del código anterior es la forma en la que hemos construido la lista con los nombres de las sesiones:

```
let sesiones = [ gnome, kde, xfce, xmonad ]
sesionesTxt <- mapM (?: text) sesiones
```

Observa que hemos utilizado `mapM` en lugar de `map`. ¿Por qué no podemos usar `map`? Si utilizáramos `map` obtendríamos una lista de acciones o valores impuros:

```
map (?: text) sesiones :: [IO String]
```

pero a la izquierda de `<-` debe aparecer una expresión de tipo acción. Una posibilidad sería utilizar `sequence`:

```
sequence :: [IO a] -> IO [a]
```

de donde obtendríamos:

```
sesionesTxt <- (sequence . map (?: text)) sesiones
```

La composición `sequence . map f` es tan habitual que está definida como `mapM`:[3]

```
mapM :: (a -> IO b) -> [a] -> IO [b]
```

Ejercicio 14. Otra forma de obtener la sesión seleccionada es aplicar mediante `mapM` la sección `(:? checked)` a la lista `sesiones`. De esta forma obtenemos una lista de Booleanos en la que sólo uno será `True`. Modifica el código anterior para aplicar esta idea.

8. Eligiendo el idioma: listas de valores

Vamos a ampliar nuestra ventana de *login* para que el usuario pueda escoger el lenguaje en que desea que se desarrolle la misma. En esta ocasión vamos a utilizar una lista de valores `ListBox a`, donde `a` es el tipo de los elementos que se pueden seleccionar en la lista (habitualmente `String`). El constructor de las listas de valores es:[4]

```
listBox :: (Container w, Data a) => [Prop (ListBox [a])] -> w -> IO (ListBox [a])
```

Las listas de valores tienen un atributo Booleano `multiple`, que indican si es posible seleccionar más de un valor. En nuestro caso, inicializaremos `multiple` a `False`, pues sólo es posible escoger un idioma. Las selección de una lista de valores se almacena en un atributo `selections`, de tipo `[Int]`. Esta lista almacena los índices de los elementos de las lista que se han seleccionado. En nuestro ejemplo, puesto que `multiple` es falso, la lista `selections` será unitaria.

[3] En realidad los tipos de `sequence` y `mapM` son un poco más complicados:

```
sequence :: Monad m =>[m a] ->m [a]
mapM :: Monad m =>(a ->m b) ->[a] ->m [b]
```

pues pueden aplicarse sobre cualquier tipo monádico `m`, y no solo sobre la mónada `IO`.

[4] En el contexto aparece `Data a`; en la práctica podemos ignorarlo. Esencialmente significa que el tipo `a` es instancia de `Read` y `Show`, y que tiene definidos un par de métodos adicionales para serializar los datos. Todos los tipos y constructores básicos son instancia de `Data`.

```
loginIdioma = gui prog
  where
    prog = do
       w <- window [ title =: "Inicio de sesión"
                   , resizable =: False
                   , on windowClose =: quit
                   ]

   -- marco de identificación de usuario

       ln <- label [ text =: "Nombre" ] w
       n  <- entry [ value =: "su nombre"
                   , font =: arial 14
                   ] w

       lc <- label [ text =: "   Clave" ] w
       c  <- entry [ value =: "su clave"
                   , font =: arial 14
                   , password =: True
                   ] w

       ocultar <- checkButton [ text =: "ocultar"
                              , checked =: True
                              , on action =: (c !: password <: not)
                              ] w

       fusr <- frame [ relief =: Groove
                     , layout =: west (ln <.< n) ^.^
                                 west (lc <.< c <.< ocultar)
                     ] w

    -- marco de tipo de sesión

       gnome  <- radioButton [ text=: "Gnome" ] w
       kde    <- radioButton [ text=: "KDE", enabled =: False ] w
       xfce   <- radioButton [ text=: "Xfce" ] w
       xmonad <- radioButton [ text=: "Xmonad", checked =: True ] w

       let sesiones = [ gnome, kde, xfce, xmonad ]
       sesionesTxt <- mapM (?: text) sesiones

       grpSesiones <- radioGroup [ radioButtons =: sesiones ]

       fses <- frame [ relief =: Groove
                     , layout =: west gnome  ^.^
                                 west kde    ^.^
                                 west xfce   ^.^
                                 west xmonad
                     ] w
```

```
-- marco de lenguaje

    let lenguajes = [ "Español", "Inglés", "Francés", "Alemán"]

    lengs <- listBox [ values =: lenguajes
                     , multiple =: False
                     , selections =: [0]
                     , size =: sz 15 4
                     ] w

    fleng <- frame [ relief =: Groove
                   , layout =: lengs
                   ] w

-- marco de botonera

    iniciar <- button [ text =: "Iniciar"
                      , on action =: do
                           imprime n c
                           numSesion <- grpSesiones ?: selection
                           putStrLn ("sesion: " ++ sesionesTxt  !! numSesion)
                           [numIdioma] <- lengs ?: selections
                           putStrLn ("idioma: " ++ lenguajes !! numIdioma)
                           quit
                      ] w

    limpiar <- button [text =: "Limpiar"
                      , on action =: do
                           limpia n c
                           xmonad !: checked =: True
                           lengs !: selections =: [0]
                      ] w

    cancelar <- button [text =: "Cancelar"
                       , on action =: quit
                       ] w

    fbot <- frame [layout =: iniciar   <.<
                             hSpace 20 <.<
                             limpiar   <.<
                             hSpace 20 <.<
                             cancelar
                  ] w

-- disposición de los marcos

    w!:layout =:   fusr ^.^
                   fses <.< hSpace 40 <.< fleng ^.^
```

```
                        fbot
```

El problema de la solución anterior es que la lista de valores aparece completamente desplegada (`size`). Si el número de valores es pequeño, esto no presenta un problema. Si el número de valores es elevado, podemos añadir barras de desplazamiento a la lista, dado que es instancia de la clase `Scrollable`. El siguiente código muestra las modificaciones que hay que hacer para asociar una barra de desplazamiento vertical `desp` (`vScrollbar`) a la lista `lengs`.

```
let lenguajes = [ "Español", "Inglés", "Francés", "Alemán"
                , "Italiano", "Portugés", "Holandés", "Danés"
                , "Sueco", "Noruego", "Finlandés"
                ]

lengs <- listBox [ values =: lenguajes
                 , multiple =: False
                 , selections =: [0]
                 , size =: sz 15 7
                 ] w

desp <- vScrollbar [] lengs

fleng <- frame [ relief =: Groove
               , layout =: lengs <|< desp
               ] w
```

9. Interfaz de un cliente de correo: la ventana de edición

Vamos a diseñar un interfaz muy simple para un hipotético cliente de correo electrónico. En cualquier interfaz de este tipo es importante contar con una ventana de edición que permita componer el texto de nuestros correos. Obviamente, el control `Entry` no resulta adecuado, pues solo nos permite introducir una línea de texto. Para poder introducir un texto de longitud arbitraria podemos emplear un control de tipo `Edit a`, cuyo constructor es:

```
edit :: (Container w, Data a) => [Prop (Edit a)] -> w -> IO (Edit a)
```

Lo más habitual es que el tipo `a` sea `String`. El texto propiamente dicho se almacena en el atributo `value`. El siguiente código muestra el esqueleto de la interfaz:

```
correo = gui prog
  where
    prog = do
       w <- window [ title =: "Correo electrónico"
                   , on windowClose =: quit
                   ]

       para <- label [ text =: "Para :" ] w
       direccion <- entry [ value =: ""
                          , width =: 45
                          ] w
```

```
fpara <- frame [ layout =: para <.< direccion ] w

cuerpo <- edit [ value =: ""
               , font =: courier 14
               ] w

enviar <- button [ text =: "Enviar"
                 , on action =: do
                      to <- direccion ?: value
                      putStrLn ("to:" ++ to)
                      body <- cuerpo ?: value
                      putStrLn "body:"
                      putStrLn body
                      quit
                 ] w

cancelar <- button [ text =: "Cancelar"
                   , on action =: quit
                   ] w


borrar <- button [ text =: "Borrar"
                 , on action =: do
                       direccion !: value =: ""
                       cuerpo !: value =: ""
                       return ()
                 ] w

botonera <- frame [layout =: enviar <.< hSpace 20 <.<
                             borrar <.< hSpace 20 <.<
                             cancelar
                  ] w

w !: layout =: west fpara    ^.^
               cuerpo   ^.^
               botonera
return ()
```

Ejercicio 15. Las ventanas de edición cuentan con el atributo Booleano `wrap`, que controla qué debe ocurrir cuando el texto llega hasta el borde derecho de la ventana de edición. Añade a la interfaz un botón de selección que permita activar o desactivar esa opción. Añade también un par de barras de desplazamiento, una horizontal y otra vertical, a la ventana de edición.

10. Añadiendo menús y cajas de diálogo habituales

Para cear un menú nos hace falta una `MenuBar`, que es un atributo exclusivo de las ventanas (el resto de contenedores no puede tener una barra de menús):

```
menuBar :: Attr Window Menu
```

Este atributo permite asociar una barra de menú a una ventana. Para crear los menús utilizamos los constructores:

```
menu :: [Prop Menu] -> IO Menu
submenu :: [Prop Menu] -> Menu -> IO Menu
```

Los dos son muy similares, la diferencia es que el segundo es un submenú de otro menú que actúa como padre. El siguiente fragmento de código muestra como crear la estructura de una barra de menús típica:

```
barra   <- menu []
archivo <- submenu [ text =: "&Archivo" ] barra
editar  <- submenu [ text =: "&Editar"  ] barra
ayuda   <- submenu [ text =: "A&yuda"   ] barra
```

Los menús aparecerán en la barra en el orden en que han sido creados. Observa que utilizamos un `&` para indicar la tecla de atajo de cada menú (`Alt` más la tecla en cuestión). Por supuesto, los menús anteriores están vacíos. Para darles contenido utilizamos los constructores:

```
menuItem :: [Prop MenuItem] -> Menu ->IO MenuItem
menuSep :: [Prop MenuSep] -> Menu -> IO MenuSep
```

Ambos mencionan el menú al que pertenecen. Vamos a completar un par de entradas del menú `archivo`:

```
menuItem [text =: "&Abrir"  ] archivo
menuItem [text =: "&Cerrar" ] archivo
menuSep  []                   archivo
menuItem [text =: "&Salir"
         , on action =: quit
         ] archivo
```

De nuevo, el orden en que se crean las entradas de menú es significativo, y las letras precedidas de `&` actúan como atajos de teclado (sin necesidad de utilizar `Alt`). Para asociar una acción a la selección de una opción del menú utilizamos la construcción `on action =: ...`. El caso de la opción `Salir` es muy simple. Para la opción `Abrir`, vamos a utilizar una caja de diálogo predefinida:

```
openFileDialog :: Window -> IO (Maybe FilePath)
```

que muestra una caja para escoger un fichero y *quizá* devuelve su ruta. La ventana que aparece como primer argumento es la ventana padre del diálogo (normalmente la ventana principal de la aplicación) y se bloquea mientras el diálogo esté activo.

Otros diálogos comunes son las cajas de mensaje:

```
okMessageBox :: Window -> String -> String -> IO Bool
okCancelMessageBox :: Window -> String -> String -> IO Bool
yesNoMessageBox :: Window -> String -> String -> IO Bool
```

El primer argumento es la ventana padre, el segundo es el título de la caja de diálogo, y el tercero es el mensaje mostrado en la caja. Estas cajas muestras los botones adecuados, y devuelven un Booleano que indica qué botón se presionó.

El siguiente ejemplo muestra el esqueleto de una aplicación con varios menús típicos:

```
aplicacion = gui prog
 where
   prog = do
      app <- window [ title =: "Aplicación"
                    , size =: sz 500 300
                    ]
      app !: on windowClose =: quit -- cerrarApp app

      barra <- menu []
      menuArchivo app barra
      menuEditar  app barra
      menuAyuda   app barra
      app !: menuBar =: barra

      return ()


menuArchivo :: Window -> Menu -> IO Menu
menuArchivo app barra = do
   archivo <- submenu [ text =: "&Archivo" ] barra

   menuItem [text =: "&Abrir"
            , on action =: abrirFichero app
            ] archivo

   menuItem [text =: "&Cerrar"
            , enabled =: False
            ] archivo

   menuSep  []archivo

   menuItem [ text =: "&Salir"
            , on action =: cerrarApp app
            ] archivo

   return archivo

menuEditar :: Window -> Menu -> IO Menu
menuEditar _ barra = do
   editar  <- submenu [ text =: "&Editar"  ] barra
   menuItem [ text =: "&Buscar"     ] editar
   menuItem [ text =: "&Reemplazar" ] editar
   return editar

menuAyuda :: Window -> Menu -> IO Menu
menuAyuda app barra = do
   ayuda   <- submenu [ text =: "A&yuda"   ] barra
   menuItem [text =: "Acerca de"
            , on action =: acercaDe app
```

```
            ] ayuda
   return ayuda

abrirFichero :: Window -> IO ()
abrirFichero app = do
      ruta <- openFileDialog app
      procesaRuta ruta

procesaRuta  Nothing = putStrLn "no se escogió ningún fichero"
procesaRuta (Just r) = putStrLn ("se escogió el fichero " ++ r)

cerrarApp :: Window -> IO ()
cerrarApp app = do
   fin <- okCancelMessageBox app "Cerrar" "¿Está seguro?"
   if fin then quit
          else return ()

acercaDe :: Window -> IO ()
acercaDe app = do
   okMessageBox app "Acerca de..." ("Ampliación de Programación\n" ++
                                    "GUIs en Haskell con AP.GUI")
   return ()
```

Ejercicio 16. En algunas aplicaciones es posible acceder a todo el menú de barra como si se tratase de un menú contextual, presionando el botón derecho del ratón. Emplea `popupAt` para añadir tal opción al anterior esqueleto de aplicación.

11. Cuenta atrás: temporizadores

A veces resulta útil añadir temporizadores a un GUI. Un temporizador es un evento que se dispara cada cierto tiempo. Los temporizadores se crean con el constructor `timer`:

```
timer :: [Prop Timer] -> IO Timer
```

El atributo más importante de un temporizador es `interval`, que indica cada cuántos milisegundos se activa. Cada vez que se produce una activación del temporizador, se lanza su manejador de eventos `action`. Hay un detalle importante a tener en cuenta: si un temporizador se activa cada `k` milisegundos, también se activa en el milisegundo cero, es decir, en el momento de su creación. El siguente ejemplo muestra una ventana de cuenta atrás:

```
cuentaAtras :: Int -> IO ()
cuentaAtras n = gui prog
  where prog = do
        w <- window [ title =: "Cuenta atrás"
                    , on windowClose =: quit
                    ]

        contador <- entry [ value =: n + 1
                          , font =: (bold . arial) 60
                          , justify =: JustifyCenter
```

```
                    ] w


segundo <- timer [ interval =: 1000
                 , on action =: do
                       contador !:  value <: flip (-) 1
                       valor <- contador ?: value
                       if valor == 0 then contador !: enabled =: False
                                     else return () -- o quit...
                 ]

w !: layout =: contador
return ()
```

Observa que hemos añadido una unidad a la cantidad total de espera `n+1`, para compensar la primera activación del temporizador.

Ejercicio 17. Nuestra ventana de cuenta atrás utiliza un control `Entry` para visualizar el contador. El problema de esta solución es que un control `Entry` es editable, por lo que el usuario puede alterar la cuenta atrás. Para evitar esto, es preferible emplear un control `Label`. El inconveniente del control `Label` es que su valor es un dato de tipo `String`, sobre el que no se puede realizar operaciones aritméticas directas. Una posibilidad es realizar conversiones mediante `read` y `show`. La otra posibilidad es utilizar un control `Var`, que no es más que una variable mutable. Utiliza ambas técnicas para reemplazar el control `Entry` por un control `Label`. Añade, además, botones para detener, continuar e inicializar la cuenta.

12. Ejercicios

Ejercicio 18. Diseña una interfaz gráfica lo más completa posible para alguno de los ejercicios del tema anterior (excepto el del juego del Nim). Por ejemplo, puedes diseñar una interfaz para obtener y mostrar el histograma de frecuencias de palabras de un texto, para cifrar o descifrar un texto, etc. Si realizas el ejercicio del histograma, puedes visualizarlo empleando un lienzo (control `Canvas`).

Ejercicio 19. Diseña un programa interactivo para jugar a algún juego de estrategia. Puede ser el Nim, las tres en raya o cualquier otro similar. El programa debe permitir al menos que se enfrenten dos personas por turnos (habilitando y deshabilitando los controles alternativamente). También puedes implementar una estrategia de juego para competir contra la máquina.

Ejercicio 20. Diseña un sistema de notas similar a *Tomboy*[5]. Aunque el sistema de notas no tiene que incluir todas las opciones de *Tomboy*, debe constar de una ventana principal en que aparezca un índice con las notas. El contenido de cada nota puede o no mostrarse en su propia ventana. Cada nota puede tener unos atributos (color de fondo, tipo de letra, etc.) diferentes, que se deben poder configurar. Añade alguna funcionalidad interesante, como por ejemplo:

- Las notas pueden salvarse a un fichero e importarse del mismo.
- Las notas pueden tener asociado un temporizador. Los atributos de la nota pueden modificarse cuando quede poco para que el temporizador expire, y volver a cambiarse cuando ha expirado.

[5] http://projects.gnome.org/tomboy/index.html

Ejercicio 21. El juego de la vida de John Horton Conway es un autómata celular bidimensional; es decir, una matriz de $m \times n$ células. Cada una de estas células puede estar viva o muerta. El estado de cada célula depende del estado de ella misma y del de sus vecinas. En el campus virtual aparece un fichero `life.pdf` con una explicación del juego y una implementación en Haskell. También tienes disponible el código fuente del juego en un fichero de Haskell ilustrado (`life.lhs`). El código que se muestra incluye una parte pura que implementa las reglas del juego, y otra impura que implementa una interfaz en modo texto (emplea las secuencias de escape ANSI del terminal). Reemplaza la parte impura del código para utilizar una interfaz gráfica. Añade los controles y opciones que creas oportunas (detener/continuar, salvar o leer la configuración del autómata, etc.). Una posibilidad interesante consiste en codificar en cada célula su edad (el número de generaciones que lleva viva), y asociar un color diferente a cada edad. Sugerencia: utiliza el *layout* `matrix` para obtener la representación del autómata.