

Tp final-Programación funcional

Integrantes :Jorge Gómez (legajo 52055), Fernando Bejarano (legajo 52043)

Profesores:Pablo Martínez López, Valeria Pennella.

Contents

1	Introducción	3
2	Programación Funcional	3
3	Desiciones de diseño.	3
4	Interfaz gráfica	3
5	Funcionalidades.	5
5.1	Abrir archivo	5
5.2	Guardar un archivo	5
5.3	Nuevo archivo.	5
5.4	Copiar	5
5.5	Pegar	5
5.6	Resaltado de sintaxis de haskell	5
5.7	Corrector ortográfico	8
5.8	Colapsar definiciones.	10
5.9	Macheo de paréntesis y de llaves	12
5.10	Búsqueda de palabras	12
6	Códigos y módulos implementados	14

7 Código Relevante	14
7.1 Main (Main2.hs)	14
7.2 CopyFromClipboard (ClipboardModule.hs)	14
7.3 pasteFromClipboard (ClipboardModule.hs)	15
7.4 readFileIntoTextView (FileModule.hs).	15
7.5 writeFileFromTextView (FileModule.hs)	16
7.6 Funciones de SyntaxHighlightModule.hs	16
7.7 Funciones de FoldingModule.hs	16
7.8 markWord (SearchModule).	19
7.9 Funciones de SpellingModule.hs	22
8 Dependencias del proyecto	25
9 Compilación y ejecución del proyecto	25
10 Conclusiones	26
11 Bibliografía	26

1 Introducción

Este trabajo consiste en plasmar los conocimientos aprendidos durante la materia Programación Funcional. Como tema para el desarrollo del mismo se decidió elegir Interfaces Gráficas en el lenguaje aprendido en la materia, Haskell. Para cumplir con el objetivo del trabajo, se decidió implementar un editor de texto plano utilizando la interfaz gráfica GTK. En el presente informe, se detalla la implementación del mismo.

2 Programación Funcional

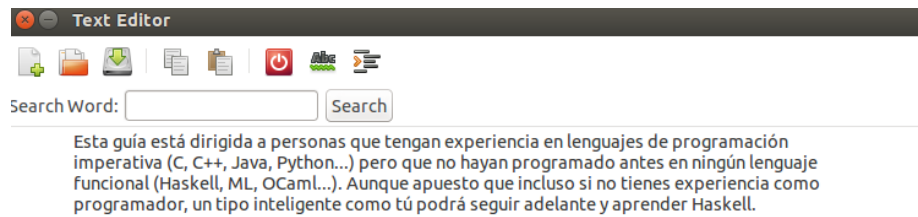
La programación funcional es un estilo de programación en el cuál las funciones no tienen efectos colaterales, es decir, solo realizan un cálculo y retornan un resultado. Esto es conocido como transparencia referencial y los lenguajes orientados a objetos e imperativos no lo cumplen. De esta forma, los resultados de una función se pueden predecir fácilmente, y se pueden encontrar funciones equivalentes.

3 Decisiones de diseño.

Se decidió emplear el lenguaje Haskell ya que fue el language funcional visto en la materia, y además ya que el mismo es compatible con la librería GTK. Para realizar la implementación de este trabajo se analizaron las librerías gráficas GTK2HS, WxHaskell y QtHaskell [5]. De estas librerías se eligió la librería GTK2HS ya que posee una excelente documentación en comparación con las otras. Se toma como base el tutorial [1]. Entre las funcionalidades que ofrece este editor, se encuentra el resaltado de texto con sintaxis Haskell y resaltado de paréntesis para hacer de esta una herramienta útil de programación en Haskell. Otras de las funcionalidades que ofrece es la corrección ortográfica; para esto se empleó una librería que utilizando Haskell invoca a la librería ortográfica Aspell de Linux; se decidió emplear Aspell porque es una de las librerías más reconocidas en Linux (de hecho viene instalada en algunas distribuciones como Ubuntu). Respecto al código, se decidió separarlo en módulos según cada funcionalidad para facilitar la comprensión del código y reutilizarlo en múltiples partes.

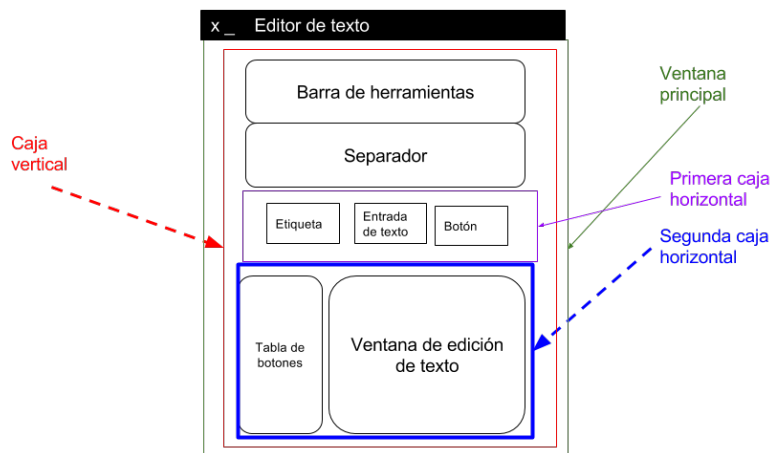
4 Interfaz gráfica

En la parte superior de la ventana de la aplicación se encuentra una barra de herramientas con los botones que proveen las funcionalidades. Los iconos de dichos botones son los que vienen por defecto en la librería GTK [7].



El texto del documento se carga con letra negra en un TextView [8] .

En el siguiente diagrama se puede apreciar un esquema sobre la implementación de la interfaz gráfica. Dentro de la ventana principal se ubica una caja vertical que contiene los principales elementos gráficos ordenados en forma vertical: la barra de herramientas, una caja horizontal, un separador, y una segunda caja horizontal. La primera caja horizontal contiene los elementos visuales que se utilizan para la funcionalidad de búsqueda de texto alineados de forma horizontal, estos son: una etiqueta (para indicar la funcionalidad), una caja que permite ingresar el texto a buscar, y un botón que activa la búsqueda. La segunda caja horizontal contiene dos elementos alineados en forma horizontal: la tabla de botones para la función de colapsado de código, y la ventana de edición de texto (TextView).



5 Funcionalidades.

5.1 Abrir archivo

Para abrir un archivo se debe tocar el botón correspondiente en la barra de herramientas. Al presionarlo, se abre una ventana de diálogo que permite elegir el archivo que se desea abrir. Una vez seleccionado el archivo, se carga el contenido del mismo en la ventana de edición de texto.

5.2 Guardar un archivo

Para utilizar esta funcionalidad se debe presionar el correspondiente botón en la barra de herramientas. Al presionarlo se abre una ventana que permite elegir el nombre y la ubicación del archivo que se desea guardar. Luego de confirmar estos datos, se guarda el contenido de la ventana de edición de texto (TextView) en un archivo.

Internamente se extrae el texto de la ventana de edición, y se emplea la función “writeFile” del módulo “System.IO” de Haskell para grabar este texto en archivo con el nombre y ubicación indicados.

5.3 Nuevo archivo.

Se borra el contenido del buffer ([10]) de la ventana de edición de texto ([8]). También se eliminan los botones del colapsado de código que hayan quedado del archivo que se tenía abierto.

5.4 Copiar

Se copia en el clipboard, lo que se haya seleccionado de la ventana principal de edición de texto.

5.5 Pegar

Se copia el contenido del clipboard en la posición del cursor en la ventana de edición de texto.

5.6 Resaltado de sintaxis de haskell

La idea es que el resaltado de sintaxis se vea forma similar a como lo realiza el editor de texto “gedit”, de manera que se asigne un color representativo a cada elemento de la sintaxis:

Text Editor

Search Word: Search

```
module FileModule where

import Graphics.UI.Gtk
import Control.Monad.IO.Class
import System.IO
import SyntaxHighlightModule
import FoldingModule

--definición de funciones

--recibe el string del nombre del archivo y el textview.
--No retorna nada. Inserta el texto del archivo en el buffer del textview.
readFileIntoTextView:: FilePath -> TextView -> Table -> IO ()
readFileIntoTextView fileName txtView table =
    do putStrLn ("Opening file: " ++ fileName)
       handle <- openFile fileName ReadMode
       contents <- hGetContents handle
       txtBuffer <- textViewGetBuffer txtView
       textBufferSetText txtBuffer contents
       --putStr contents
       hClose handle
       clearButtons table --se borran los botones para colapsar código del archivo anterior
       highlightSyntax txtView table
       return ()

--recibe el string del nombre del archivo y el textview.
--No retorna nada. Inserta el texto del buffer del textview al archivo.
writeFileFromTextView:: FilePath -> TextView -> IO ()
writeFileFromTextView fileName txtView =
    do putStrLn ("Saving file: " ++ fileName)

       txtBuffer <- textViewGetBuffer txtView
       start <- textBufferGetStartIter txtBuffer
       end <- textBufferGetEndIter txtBuffer
       contents <- textBufferGetText txtBuffer start end False
       writeFile fileName contents
       return ()

--función que se emplea para la opción de "nuevo archivo"
```

Los colores que se emplean sobre la fuente de las letras para representar elementos de la sintaxis son los siguientes:

- nombre de función: negro en cursiva.
- comentario: azul.
- tipo de dato y constructores: verde.
- las palabras reservadas (por ejemplo “data”): marrón. La lista de estas palabras se obtuvo de [4] y es la siguiente: “case”, “class”, “data”, “deriving”, “do”, “else”, “if”, “import”, “in”, “infix”, “infixl”, “infixr”, “instance”, “let”, “of”, “module”, “newtype”, “then”, “type”, “where”.
- el resto: en negro sin subrayado ni cursiva.

Esta funcionalidad se realiza en forma automática cuando se abre un archivo. En caso de que la sintaxis no sea válida, no se la resalta. En forma adicional, esta funcionalidad se puede activar con un botón en la barra de herramientas. Si luego de haberse realizado el resaltado de la sintaxis se realiza algún cambio sobre el texto, es necesario volver a presionar este botón para que se ajuste el resaltado al nuevo texto que se tiene.

En cuanto al código implementado, el módulo que se utiliza para realizar el parseo de la sintaxis de Haskell es “Language.Haskell.Parser” [9]; en particular se emplea la función “parseModule”. Si se obtiene un resultado de tipo “ParseOk” (el parseo del texto fue exitoso ya que presenta sintaxis de Haskell válida) se resalta gráficamente el código. Si se obtiene un resultado de tipo “ParseFailed” no se resalta el código.

En caso del que la sintaxis sea válida, dentro del resultado “ParseOk” se obtiene un valor tipo “HsModule” el cual representa un módulo de código fuente. El quinto parámetro que recibe el constructor de un “HsModule” es de tipo “[HsDecl]”; esto último representa un arreglo de declaraciones de Haskell. Una declaración de Haskell (valor tipo “HsDecl”) puede ser tipo “HsTypeSig”, “HsDataDecl”, o “HsFunBind” entre otros tipos.

El tipo “HsTypeSig” representa una declaración de tipos o “signature” de una función. Contiene la ubicación de dicha declaración en el texto y el nombre de la función; estos datos se emplean para resaltar el nombre de la función. Adicionalmente, el tipo “HsTypeSig” contiene un valor tipo “HsQualType” el cual representa al conjunto de tipos de datos que se declaran en el signature de la función (tipos de los parámetros y del valor de retorno); este último dato se emplea para resaltar los tipos que se declaran en el signature.

Por otra parte, el tipo “HsDataDecl” representa la declaración de un nuevo tipo de datos mediante el empleo de la palabra reservada “data”. Este tipo contiene el nombre del nuevo tipo junto con la ubicación de dicha declaración en el texto; esto se emplea para resaltar el nombre de dicho tipo. Adicionalmente, el constructor del tipo “HsDataDecl” contiene una lista de valores tipo “HsConDecl”

que representan una lista de los distintos constructores que posee el tipo de dato que se esta definiendo en sintaxis de Haskell.

Cada valor de tipo “HsConDecl” contiene el nombre y la ubicación de cada constructor (del nuevo tipo de datos) en el texto. Además, contiene una lista de los tipos que recibe cada constructor. Tanto el nombre del constructor como los tipos de datos que recibe cada constructor se resaltan del color seleccionado para los tipos, empleando los datos mencionados.

También se tiene el tipo “HsFunBind”, el cual representa a un conjunto de definiciones de una función (polimorfismo paramétrico). Cada definición de una función se representa con el tipo “HsMatch”; este último tipo contiene el nombre de la función que se esta declarando y su ubicación. Con estos datos, se procede a resaltar el nombre de la función que se declara.

En cuanto a las palabras reservadas, se busca cada una en todo el texto. En caso de que se encuentre una ocurrencia de dicha palabra, se la resalta. Este resaltado de palabras reservadas solo se realiza en caso de que el texto presente sintaxis de Haskell válida.

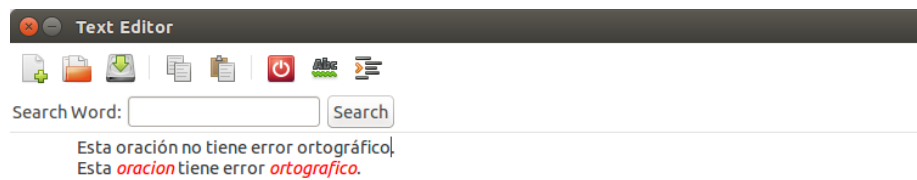
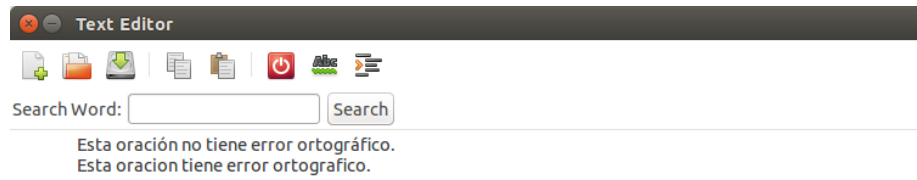
Por último, en cuando al resaltado de los comentarios, el mismo se realiza en caso de que haya sintaxis válida de Haskell. Para lograr este resaltado, se buscan ocurrencias en el texto de dos guiones seguidos (“-”); cuando esto ocurre, se marca como comentario a todo el texto que se encuentre desde los guiones hasta el final de la línea.

5.7 Corrector ortográfico

Para detectar las palabras mal escritas, se emplea la librería Aspell para Haskell (Haspell) [6]. Esta librería indica como incorrectas a aquellas palabras que no se encuentren en el diccionario que se esta empleando; la implementación hecha para este trabajo ofrece soporte para el diccionario español. Cuando se detecta una palabra mal escrita, se le asigna color de letra rojo con *itálica*.

Por ejemplo, dado el siguiente texto con errores ortográficos la aplicación se ve de la siguiente manera antes de tocar el botón del corrector:

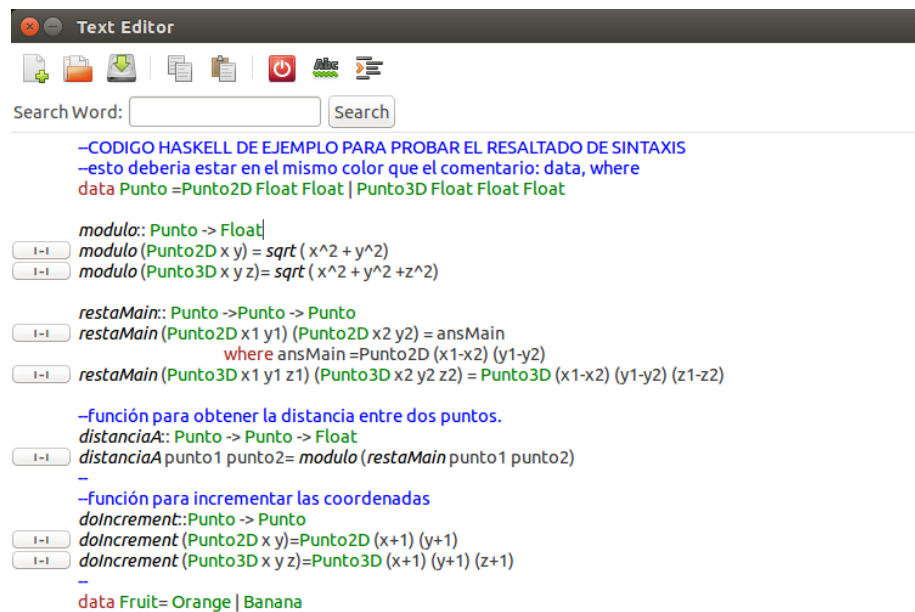
Después de tocar el botón del corrector se obtiene lo siguiente:



NOTA: se puede editar mientras se este en modo de corrección. Luego de editar, es necesario volver a activar esta funcionalidad para que se actualice el marcado de la ortografía.

5.8 Colapsar definiciones.

Cuando se tiene el texto resaltado con la sintaxis de Haskell, se ofrece la posibilidad de colapsar las definiciones de funciones de Haskell. Se muestra un botón con el símbolo “[−]” en el margen izquierdo de la línea donde esta definida la función. Esto se puede apreciar en la siguiente captura de pantalla:



Cuando se presiona dicho botón, se colapsa la definición completa de la función y solo se deja el nombre; luego de realizar esto, dicho botón queda oscurecido para indicar que la función se encuentra activada. Esto se puede ver en la siguiente captura de pantalla de la aplicación:

```
Text Editor

Search Word:  Search

--CODIGO HASKELL DE EJEMPLO PARA PROBAR EL RESALTADO DE SINTAXIS
--esto deberia estar en el mismo color que el comentario: data, where
data Punto = Punto2D Float Float | Punto3D Float Float Float

modulo: Punto -> Float
modulo (Punto2D x y) = sqrt ( x^2 + y^2)
modulo (Punto3D x y z) = sqrt ( x^2 + y^2 + z^2)

restaMain: Punto -> Punto -> Punto
restaMain
restaMain (Punto3D x1 y1 z1) (Punto3D x2 y2 z2) = Punto3D (x1-x2) (y1-y2) (z1-z2)

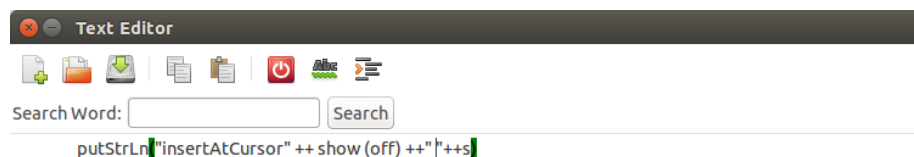
--función para obtener la distancia entre dos puntos.
distanciaA: Punto -> Punto -> Float
distanciaA punto1 punto2 = modulo (restaMain punto1 punto2)
--
--función para incrementar las coordenadas
doIncrement: Punto -> Punto
doIncrement (Punto2D x y) = Punto2D (x+1) (y+1)
doIncrement (Punto3D x y z) = Punto3D (x+1) (y+1) (z+1)
--
data Fruit = Orange | Banana
```

En caso de que se vuelva a presionar el botón, se vuelve a mostrar la definición de la función y botón vuelve a su estado anterior (no oscurecido para indicar que no esta activada esta funcionalidad).

Al presionar estos botones, si se colapsa una o más líneas debajo de una función, los botones de las funciones que se encuentran debajo de la primera deben ser movidos hacia arriba en la misma proporción de líneas que se colapsaron. Cuando se realiza el proceso inverso (se desactiva el colapsado del código sobre una función), se mueven hacia abajo los botones de las funciones que se encuentran abajo de la función a la cual se le restaura la definición.

5.9 Macheo de paréntesis y de llaves

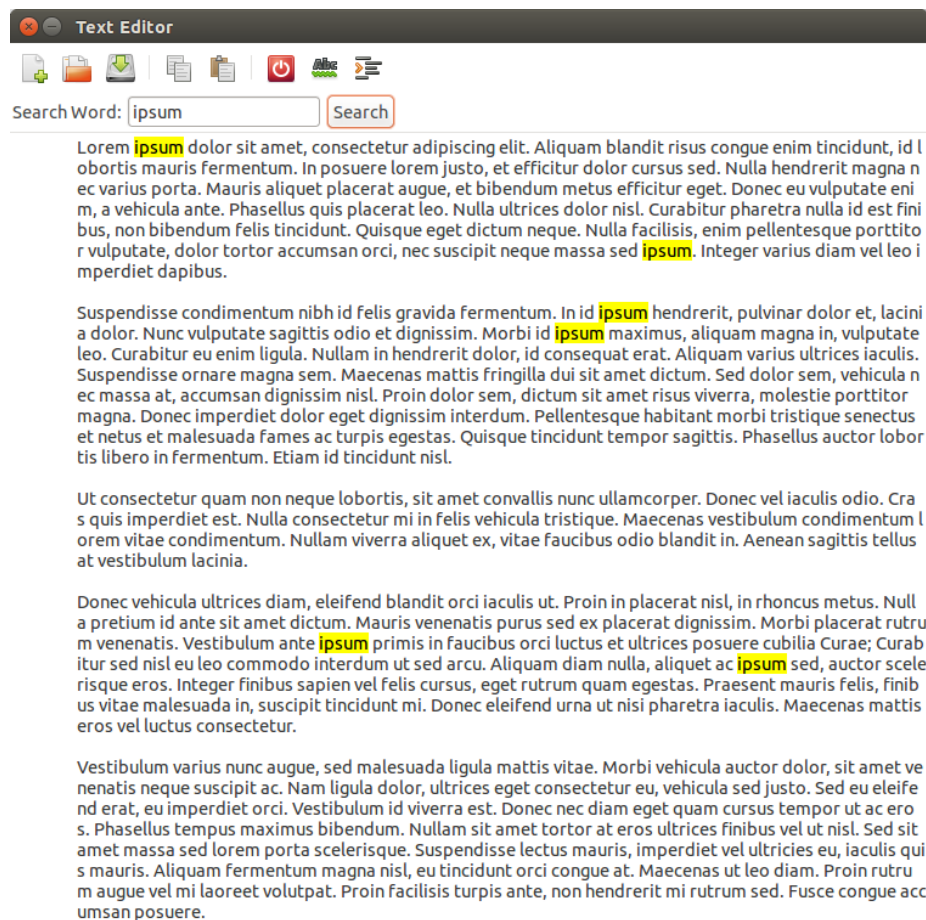
En el caso de se ubique el cursor sobre caracteres que se encuentran dentro de un par de paréntesis, se resalta dicha apertura de paréntesis junto con el paréntesis que cierra. Por otra parte, si no se cierra el paréntesis que abre, no se lo resalta. Para el resaltado, se emplea el color verde como color de fondo del símbolo a resaltar; el color de la fuente no se modifica. Esta funcionalidad se puede apreciar en el siguiente prototipo:



Debido a las limitaciones de la interfaz gráfica utilizada, no se pudo encontrar una forma de interceptar el evento luego de que se mueve el cursor en el texto, solo el evento antes de moverse. Por ese motivo, se resaltan los parentesis que contienen la posicion anterior luego de mover el cursor. Si bien, dentro de una función con mucha distancia entre los paréntesis este efecto no se puede notar, si se nota cuando la distancia entre los parentesis es muy corta.

5.10 Búsqueda de palabras

Junto con los botones mencionados anteriormente, se ofrece en la barra superior un campo en el cual se puede ingresar una cadena de caracteres a buscar. Al lado de dicho campo hay un botón que al presionarlo se resaltan con fondo amarillo aquellas cadenas que coincidan con lo buscado.



6 Códigos y módulos implementados

- Main2.hs: es el código principal de la aplicación. En el mismo se inicializa la interfaz gráfica y se inicializan las funciones que se pueden realizar con los botones de la barra de herramientas.
- ClipboardModule.hs: implementa las operaciones de copiar al clipboard y pegar desde el clipboard.
- FileModule.hs: implementa la lectura y escritura de archivos.
- FoldingModule.hs: modulo que realiza la función de colapsado de código de una función.
- SearchModule: implementa la búsqueda y el marcado de palabras.
- SpellingModule.hs: implementa la corrección ortográfica.
- SyntaxHighlightModule.hs: contiene la implementación del parseo y marcado de la sintaxis de Haskell.
- SyntaxUtilsModule.hs: contiene funciones de uso común para los módulos “SyntaxHighlightModule.hs” y “FoldingModule.hs”.
- TagsModule: modulo que implementa distintas marcas que se utilizan en varios módulos y que se pueden aplicar sobre partes del texto.

7 Código Relevante

7.1 Main (Main2.hs)

Función principal donde se inicializa la interfaz gráfica.

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

7.2 CopyFromClipboard (ClipboardModule.hs) .

Se emplea para brindar la funcionalidad de “copiar”.

```
copyFromClipboard :: ActionClass self => (self, TextView) -> IO (ConnectId self)
copyFromClipboard (a, txtview) = onActionActivate a $
    do putStrLn ("Copy to clipboard")
       readClipboard <- clipboardGet selectionPrimary
       writeClipboard <- clipboardGet selectionClipboard
       clipboardRequestText readClipboard (copyCallBack writeClipboard)
```

Se obtiene el clipboard de selección de la interfaz gráfica y el clipboard general del sistema operativo. Por último, el texto seleccionado en el clipboard de la interfaz

gráfica (texto marcado en la ventana de edición) se graba en el clipboard del sistema operativo. “selectionClipboard” es una función que devuelve clipboard general del sistema operativo y “selectionPrimary” es el clipboard de selección de la ventana de edición de texto (TextView). Con la función “copyCallback” se termina copiando el texto al clipboard del sistema operativo.

7.3 pasteFromClipboard (ClipboardModule.hs)

Se utiliza para brindar la funcionalidad de “pegar”.

```
pasteFromClipboard:: ActionClass self => (self,TextView) -> IO (ConnectId self)
pasteFromClipboard (a, txtview) = onActionActivate a $
    do putStrLn ("Paste from clipboard")
       clipboard <- clipboardGet selectionClipboard
       clipboardRequestText clipboard (pasteCallback txtview)
```

Primero se obtiene el clipboard del sistema operativo, luego se obtiene el texto que se encuentra en dicho clipboard y por último se lo copia en el buffer de la ventana de edición de texto empleando la función “pasteCallBack”; esta última se emplea en forma asincrónica a través de la función “clipboardRequest”. La función “selectionClipboard” devuelve el clipboard general del sistema operativo.

7.4 readFileIntoTextView (FileModule.hs).

Esta función se utiliza para brindar la funcionalidad de abrir un archivo.

```
readFileIntoTextView:: FilePath -> TextView->Table -> IO ()
readFileIntoTextView fileName txtView table =
    catchIOError (
        do putStrLn ("Opening file: " ++ fileName)
           handle <- openFile fileName ReadMode
           contents <- hGetContents handle
           txtBuffer <- textViewGetBuffer txtView
           textBufferSetText txtBuffer contents
           hClose handle
           clearButtons table --se borran
    los botones para colapsar código del archivo anterior
           highlightSyntax txtView table
           return ()
    ) (\x -> do
        putStrLn("INVALID FILE")
        return ())
```

Esta función recibe el string del nombre del archivo a abrir y el textview. Se emplea la función “openFile” del módulo “System.IO” para abrir el archivo en modo lectura. Como resultado de esto, se obtiene un “Handle”; mas tarde se obtiene el texto del archivo empleando la función “hGetContents” la cual recibe como parámetro el handle. Una vez que se tiene el contenido del archivo, se lo carga en el buffer de la ventana de edición de texto ([8]). Por último, se cierra el handle. En caso de que se obtenga una excepción (por ejemplo si el archivo es inválido), se la captura para que la aplicación no falle y no se realiza ningún cambio en la interfaz gráfica.

7.5 writeFileFromTextView (FileModule.hs)

La siguiente función se emplea para brindar la funcionalidad de guardar un archivo.

```
--recibe el string del nombre del archivo y el textview.
--No retorna nada. Inserta el texto del buffer del textview al archivo.
writeFileFromTextView :: FilePath -> TextView -> IO ()
writeFileFromTextView fileName txtView =
    do putStrLn ("Saving file: " ++ fileName)

    txtBuffer <- textViewGetBuffer txtView
    start <- textBufferGetStartIter txtBuffer
    end <- textBufferGetEndIter txtBuffer
    contents <- textBufferGetText txtBuffer start end False
    writeFile fileName contents
    return ()
```

Recibe el string del nombre con el cual se desea guardar el archivo y el textview; luego copia el contenido completo del textview (dejando de lado las etiquetas) en un archivo con el nombre indicado.

7.6 Funciones de SyntaxHighlightModule.hs

7.7 Funciones de FoldingModule.hs

Las funciones de este módulo son las que se emplean para brindar el colapsado de código. Esta funcionalidad empieza en la función “processFolding” con el resultado del parseo del texto obtenida en el resaltado de la sintaxis de Haskell; es decir que se recibe un valor tipo “HsModule”.

```
processFolding :: HsModule -> TextBuffer -> Table -> IO ()
processFolding (HsModule _ _ _ _ hsDecl) buffer table = do
```



```

clearButtons table
processHsDecl buffer hsDecl table
widgetShowAll table
putStrLn "[processFolding] buttons done"

```

En forma similar al proceso de marcado de sintaxis (ver análisis hecho en la sección “Funciones de SyntaxHighlightModule.hs”), se procesa los distintos valores contenidos empezando en la función “processHsDecl” hasta llegar a los valores tipo “HsMatch”. Cuando esto último ocurre, en la función “processHsMatch” se considera al final del nombre de la función como posición inicial del texto a ocultar, y como posición final se toma el comienzo de la siguiente declaración de función; en caso de que no exista una definición de función siguiente, se considera como posición final al comienzo de cualquier declaración de Haskell que se encuentre luego de la definición de la función actual.

```

--Primera lista contiene las siguientes declaraciones de Haskell.

--Segunda lista tiene como primer elemento a la declaración de la actual función;
--los siguientes elementos son las siguientes funciones.
processHsMatch :: [HsDecl] -> [HsMatch] -> TextBuffer->Table->IO ()
processHsMatch xs (y:ys) buffer table = do
    --iterador al comienzo del código a ocultar
    start<-getNameEndIter y buffer

    (end,nextLine)<- if (null ys)
        then
            --comienzo siguiente declaración
            getStartIter xs buffer
        else
            --comienzo de la siguiente función
            getNameStartIter ys buffer
    tags <- textBufferGetTagTable buffer
    tag<-invisibleTag
    textTagTableAdd tags tag

    startOffset<-textIterGetOffset start
    endOffset<-textIterGetOffset end

    --se crea el boton para colapsar el código de dicha función
    let currentLine=getRow y
    let row=getRow y
    button<-createButton start end tag buffer
    tableAttachDefaults table button 0 1 (row-1) row

    --llamada recursiva.

```

```

processHsMatch xs ys buffer table

onToggled button (buttonSwitch button buffer
  tag start end table (currentLine - nextLine ))

processHsMatch _ _ _ _ =return ()

```

Una vez que se tiene la posición inicial (iterador guardado en la variable “start”) y final (iterador guardado en la variable “end”) del texto de la definición de función a ocultar, se crea un botón que se encuentre en la misma línea respecto de la definición de función. A dicho botón se lo configura para que al ser presionado, invoque una función (llamada “buttonSwitch” en el modulo “FoldingModule.hs”) que realiza el ocultamiento del texto entre las posiciones que se tienen (posición inicial y final) y además invoca otra función que lleva a cabo el reordenamiento de los otros botones.

Para ocultar el texto, se emplean etiquetas en el buffer de la ventana de edición que lo vuelven “invisible” [11], aunque en realidad no se lo borra. Estas etiquetas son creadas con la función “invisibleTag”. Para volver a mostrar este texto, simplemente se quitan estas marcas y luego el texto vuelve a ser visible en la la ventana de edición.

El comienzo de la siguiente declaración se extrae del valor tipo SrcLoc que contine dicha declaración. Esto se encuentra implementado en las funciones getStartIter y getStartIterRec:

```

getStartIter::[HsDecl]->TextBuffer->IO (TextIter,Int)
getStartIter [] buffer=do
    iter<- textBufferGetEndIter buffer
    return (iter,0)
getStartIter (x:xs) buffer =getStartIterHsDecl x buffer

--
getStartIterHsDecl::HsDecl->TextBuffer->IO (TextIter,Int)
getStartIterHsDecl (HsTypeDecl srcLoc _ _ _) buffer=getIterForSrcLoc srcLoc buffer
getStartIterHsDecl (HsDataDecl srcLoc _ _ _ _) buffer=getIterForSrcLoc srcLoc buffer
getStartIterHsDecl (HsInfixDecl srcLoc _ _ _) buffer=getIterForSrcLoc srcLoc buffer
getStartIterHsDecl (HsNewTypeDecl srcLoc _ _ _ _ _) buffer=getIterForSrcLoc srcLoc buffer
getStartIterHsDecl (HsClassDecl srcLoc _ _ _ _ _) buffer=getIterForSrcLoc srcLoc buffer
getStartIterHsDecl (HsInstDecl srcLoc _ _ _ _ _) buffer=getIterForSrcLoc srcLoc buffer
getStartIterHsDecl (HsDefaultDecl srcLoc _ ) buffer=getIterForSrcLoc srcLoc buffer
getStartIterHsDecl (HsTypeSig srcLoc _ _) buffer=getIterForSrcLoc srcLoc buffer
getStartIterHsDecl (HsFunBind hsMatch ) buffer=getNameStartIter hsMatch buffer
getStartIterHsDecl (HsPatBind srcLoc _ _ _ ) buffer=getIterForSrcLoc srcLoc buffer

```

```
getStartIterHsDecl (HsForeignImport srcLoc _ _ _ _ ) buffer=getIterForSrcLoc srcLoc buffer
getStartIterHsDecl (HsForeignExport srcLoc _ _ _ _ ) buffer=getIterForSrcLoc srcLoc buffer
```

El iterador que apunta a la posición en el texto en la cual comienza la siguiente función se obtiene con la función “getNameStartIter”:

```
--Retorna el iterador que apunta al comienzo del nombre de la definición de función
getNameStartIter::[HsMatch]->TextBuffer->IO (TextIter,Int)
getNameStartIter ((HsMatch srcLoc _ _ _ _):ys) buffer=getIterForSrcLoc srcLoc buffer
```

Tanto la función “getStartIterHsDecl” como la función “getNameStartIter” terminan utilizando la función getIterForSrcLoc, la cual es la que crea el iterador a partir del valor tipo SrcLoc. Ésta última función, extrae del valor tipo “SrcLoc” el número de columna y el número de fila, y con esos datos crea un iterador apuntando a la correspondiente posición en el texto.

```
getIterForSrcLoc::SrcLoc->TextBuffer->IO (TextIter,Int)
getIterForSrcLoc srcLoc buffer=do
    let line=(srcLine srcLoc)-1
    let column=(srcColumn srcLoc)
    iter<-textBufferGetIterAtLine buffer line
    textIterForwardChars iter (column -1 )
    return (iter,line)
```

7.8 markWord (SearchModule).

Esta función se emplea para marcar una determinada palabra con una determinada etiqueta. Esta función es llamada en la implementación de la funcionalidad de búsqueda de palabras y en el resaltado de palabras reservadas del lenguaje Haskell.

```
--primer parámetro es el buffer
--el segundo parámetro es la etiqueta a emplear para marcar
--el tercer parámetro es la palabra que se busca.
markWord::TextBuffer->TextTag->String->IO()
markWord buffer tag name=do
    start<-textBufferGetStartIter buffer
    end<-textIterCopy start
    tags <- textBufferGetTagTable buffer
    textTagTableAdd tags tag
    markWordRec buffer name start end "" tag 0 False False
```

La anterior función llama a la función recursiva “markWordRec”, la cual es la que realiza la búsqueda y marcado de una palabra en forma recursiva.

```
markWordRec::TextBuffer->String->TextIter->TextIter->String->TextTag->Int->Bool->Bool->IO()
markWordRec buffer name start end acum tag nameOffset inOtherWord stopOnNewLine=do
  startOffset<-textIterGetOffset start
  endOffset<-textIterGetOffset end
  --posición limite
  top<-textBufferGetEndIter buffer
  topOffset<-textIterGetOffset top

  if( (not inOtherWord) && (compare name acum)== EQ)
    then do

      startOffset<-textIterGetOffset start
      endOffset<-textIterGetOffset end

      --se evitan ocurrencias parciales al marcar
      nextIter<-textIterCopy end
      textIterForwardChars nextIter 1
      currentString<-textBufferGetText buffer

      let nextChar'=currentString !! 0
      CM.when ((endOffset == topOffset)
        || DC.isSpace nextChar'
        || (nextChar' == '.')
        || (nextChar' == ':')
        || (nextChar' == ';')
        || (nextChar' == ',')
        || (nextChar' == '-')
        || (nextChar' == ']')
        || (nextChar' == '=')
      )
        (textBufferApplyTag buffer tag start end)

      --se busca siguiente ocurrencias si no
      se encuentra en el final del buffer
      if (endOffset < topOffset)
        then do
          textIterForwardChars end 1
          start<-textIterCopy end
          markWordRec buffer name
        start end "" tag 0 False stopOnNewLine
      else
        return ()
```

```

else do
  --posición caracter actual
  current<-textIterCopy end

  textIterForwardChars end 1
  currentString<-textBufferGetText

buffer current end False

  let currentChar=currentString !! 0
  --Caso en el cual se
  encuentre en el último elemento del buffer
  if (endOffset == topOffset
|| ( (compare currentChar '\n')==EQ && stopOnNewLine) )
  then return ()
  else do

    --el elemento buscado no esta en esta
    posición, se busca mas adelante
    if( (length name)<= nameOffset
||not( (compare currentString
  [(name !! nameOffset)])==EQ)
    )
    then do
      start<-textIterCopy end
      let inOtherWord=not (
        DC.isSpace currentChar
        || (currentChar == '.'')
        || (currentChar == ':'')
        || (currentChar == ';'')
        || (currentChar == ','')
        || (currentChar == '='')
        || (currentChar == '>')
        || (currentChar == '['')
        || (currentChar == '(''))
        markWordRec
      buffer name start end "" tag 0 inOtherWord stopOnNewLine

    else do
      --se obtiene el siguiente caracter
      contents <- textBufferGetText

      buffer start end False

      markWordRec buffer name start end
      contents tag (nameOffset +1) inOtherWord stopOnNewLine

```

La función “markWordRec” recibe los siguientes parámetros: + el buffer del textview. + la palabra a marcar. + un iterador que apunta al comienzo del nombre de la palabra actual en el buffer. + un iterador que apunta al final de la palabra. + la cadena leída hasta el momento en el buffer, + la etiqueta a emplear. + un contador que indica la cantidad de caracteres de la palabra actual que coinciden con la palabra buscada. + un valor booleano que indica si se está iterando sobre una palabra distinta de la que se busca + un booleano que indica si se debe parar de buscar la palabra frente a un fin de línea.

Lo primero que realiza esta función es comprobar si la cadena de caracteres acumulada coincide con la palabra buscada y además se verifica que la palabra no se encuentre dentro de otra (para evitar búsquedas parciales); si esto se cumple se procede a marcar la ocurrencia actual de la palabra y luego se realiza una llamada recursiva para seguir buscando ocurrencias.

Si el anterior caso no se dio, se analiza el siguiente carácter en el texto. Si este carácter resulta ser un salto de línea y si se realiza una búsqueda hasta el primer salto de línea, se deja de buscar la palabra. Por otro lado, si se llega a un salto de línea y se realiza una búsqueda en todo el texto (no hasta el primer salto de línea), se continúa buscando en el resto del texto.

Luego, si se llega a un carácter de puntuación o un espacio, se realiza una nueva búsqueda de la palabra.

7.9 Funciones de SpellingModule.hs

Para realizar el marcado de palabras mal escritas se llama a la función “markSpelling”, la cual a su vez llama a la función “markSpellingRec” especificándole el tipo de etiqueta con el cual se marca a las palabras.

```
markSpelling :: TextView -> IO()
markSpelling txtview = do
    txtBuffer <- textViewGetBuffer txtview

    start <- textBufferGetStartIter txtBuffer

    end <- textBufferGetEndIter txtBuffer

    --se remueven marcas previas
    textBufferRemoveAllTags txtBuffer start end

    contents <- textBufferGetText txtBuffer start end False
    tags <- textBufferGetTagTable txtBuffer
    kindaRedItalic <- textTagNew Nothing
    set kindaRedItalic [
        textTagStyle := StyleItalic,
```

```

        textTagForegroundSet := True,
        textTagForeground := ("red" :: String)
    ]
    textTagTableAdd tags kindaRedItalic
end' <- textIterCopy start
markSpellingRec contents start end' "" txtBuffer kindaRedItalic
--

```

La función “markSpellingRec” realiza el marcado en forma recursiva sobre todo el texto. En este sentido, se lee el texto hasta que se encuentra un separador de palabras. Se considera como separador de palabras a los espacios, el símbolo “.” y el punto y coma. Cuando se encuentra un separador, se analiza el texto leído hasta dicho separador; en este sentido se emplea la función auxiliar “spellCheck”. En caso de que dicha palabra no sea correcta, se procede a marcarla en la ventana de edición de texto.

```

markSpellingRec:: [Char] -> TextIter ->TextIter ->String ->TextBuffer->TextTag->IO()
markSpellingRec [] start end acum txtbuffer tag=return ()
markSpellingRec (x:xs) start end acum txtbuffer tag=do
    startOffset'<-textIterGetOffset start
    endOffset'<-textIterGetOffset end
    if (DC.isSpace x
        || (x == '.'))
        || (x == ':')
        || (x == ';'))
    then
        do
            spellPass<-spellCheck acum
            if (not spellPass)
            --hay error de ortografia
            then do
                --textIterBackwardChars end 1
                auxStart <-textIterCopy end
                startOffset<-textIterGetOffset start
                endOffset<-textIterGetOffset end
                --se marca el error
                textBufferApplyTag txtbuffer tag start end
                textIterForwardChars end 1

                markSpellingRec xs (auxStart) (end) "" txtbuffer tag

            else do
                textIterForwardChars end 1
                auxStart <-textIterCopy end

```

```

                                markSpellingRec xs auxStart end "" txtbuffer tag
else do
    textIterForwardChars end 1
    markSpellingRec xs start end (acum ++ [x]) txtbuffer tag

```

Dentro de la función “spellCheck” se obtiene un corrector ortográfico mediante la función “spellCheckerWithOptions” de la librería Aspell, y luego se termina utilizando este corrector junto con la palabra para determinar si la misma esta correctamente escrita.

```

spellCheck::String ->IO Bool
spellCheck string=do
    aux <-spellCheckerWithOptions
    [(LAO.Dictionary bytestringLang),LAO.Encoding LAO.Latin1]
    let checker = unpack' aux
    let bytestringInput =BC.pack string
    --se termina llamando al corrector ortografico
    return (check checker bytestringInput)

```

Para marcar las palabras mal escritas se utilizan etiquetas en la ventana de edición de texto (TextView) del aplicativo, asignándoles color de letra rojo con itálica. Se emplean iteradores que poseen la posición de comienzo y de fin de la palabra que se analiza en el buffer de la ventana de edición de texto; estos iteradores se especifican junto con la etiqueta para realizar el marcado.

```

markSpellingRec:: [Char] -> TextIter ->TextIter ->String ->TextBuffer->TextTag->IO()
markSpellingRec [] start end acum txtbuffer tag=return ()
markSpellingRec (x:xs) start end acum txtbuffer tag=do
    startOffset'<-textIterGetOffset start
    endOffset'<-textIterGetOffset end
    if (DC.isSpace x || (x == ' ') || (x == ':') || (x == ';'))
    then
        do
            spellPass<-spellCheck acum
            if (not spellPass)
            --hay error de ortografia
            then do
                --textIterBackwardChars end 1
                auxStart <-textIterCopy end
                startOffset<-textIterGetOffset start
                endOffset<-textIterGetOffset end

```



```

--se marca el error
textBufferApplyTag txtbuffer tag start end
textIterForwardChars end 1

markSpellingRec xs (auxStart) (end) "" txtbuffer tag

else do
    textIterForwardChars end 1
    auxStart <-textIterCopy end
    markSpellingRec xs auxStart end "" txtbuffer tag
else do
    textIterForwardChars end 1
    markSpellingRec xs start end (acum ++ [x]) txtbuffer tag

```

8 Dependencias del proyecto

El proyecto depende de las siguientes programas y librerías:

- ghc
- gtk
- aspell
- aspell-es
- libaspell-dev
- haspell
- happy
- haskell-src

Las mismas se pueden instalar utilizando el script para Linux “setupEnvironment.sh” que se encuentra junto con los archivos del proyecto. El mismo fue utilizado exitosamente en la distribución de linux “Ubuntu 14.04” .

9 Compilación y ejecución del proyecto

Antes de compilar el programa, se necesitan tener instaladas las dependencias descriptas en el punto anterior. Para compilar el programa, se utiliza el compilador de Haskell “ghc” (“Glasgow Haskell compiler”). En el script de Linux “correr.sh” se puede ver como se compila y se ejecuta el programa. En el mismo, el ejecutable que se obtienen se llama “app”.

10 Conclusiones

Si bien la librería gráfica GTK ofrece una completa documentación, presentó múltiples problemas y limitaciones:

- la documentación carece de ejemplos.
- es imperativo.
- resulta incómodo para el manejo de interfaces gráficas con cierta complejidad por la cantidad de líneas que se necesitan en comparación con otros lenguajes.
- carece del manejo de ciertos eventos.
- es complejo de implementar el desplazamiento conjunto entre la ventana de texto y la tabla de botones.

Por otra parte, el lenguaje Haskell es muy práctico para realizar funciones matemáticas, manejo de tipo, recursividades y algoritmos. Ya que las herramientas de curificación, pattern matching y las características propias de un lenguaje funcional facilitan el desarrollo de la aplicación. Además resulta muy conveniente la separación que realiza Haskell entre lenguaje funcional puro y lenguaje impuro (con operaciones de IO).

11 Bibliografía

- [1] <http://www.muitovar.com/gtk2hs/index.html>
- [2] Bryan O'Sullivan, Don Stewart, and John Goerzen. Real World Haskell. O'Reilly, First Edition, 2009.
- [3] Miran Lipovaca. Learn you a Haskell for great good. No starch press, 2011.
- [4] <http://blog.codeslower.com/static/CheatSheet.pdf>
- [5] Gideon Sireling. Graphical user interfaces in Haskell. 2011.
- [6] Haspell: Haskell bindings to Aspell. <https://hackage.haskell.org/package/haspell-1.1.0/docs/doc-index.html>
- [7] Stock Items. The gtk Class Reference. <http://www.pygtk.org/pygtk2reference/gtk-stock-items.html>
- [8] TextView. <http://projects.haskell.org/gtk2hs/docs/devel/Graphics-UI-Gtk-Multiline-TextView.html>
- [9] Haskell parser. <https://hackage.haskell.org/package/haskell-src-1.0.2.0/docs/Language-Haskell-Parser.html>
- [10] Textbuffer. <http://projects.haskell.org/gtk2hs/docs/devel/Graphics-UI-Gtk-Multiline-TextBuffer.html>
- [11] Text tag invisible. <http://projects.haskell.org/gtk2hs/docs/devel/Graphics-UI-Gtk-Multiline-TextTag.html#v%3AtextTagInvisible>