

Tp final-Programación funcional

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

Profesores:Pablo Martínez López, Valeria Pennella.

Contents

Introducción	2
Programación Funcional	2
Interfaz gráfica	2
Funcionalidades.	3
Funcionalidades básicas	3
Resaltado de sintaxis de haskell	3
Corrector ortográfico	6
Abrir archivo	8
Guardar un archivo	8
Nuevo archivo.	8
Copiar	8
Pegar	9
Colapsar definiciones.	9
Macheo de paréntesis y de llaves	11
Búsqueda de palabras	12
Códigos y módulos implementados	13
Código Relevante	13
Dependencias del proyecto	13
Compilación y ejecución del proyecto	14
Bibliografía	14

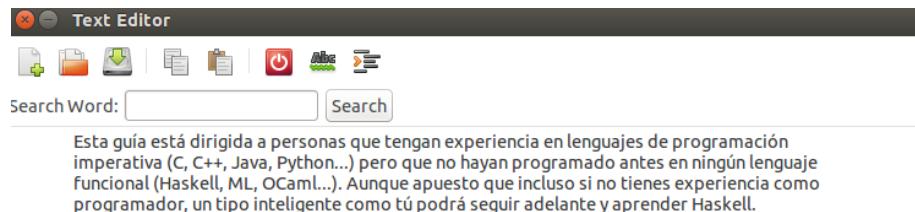
Introducción

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 utilizando la interfaz gráfica GTK. En el presente informe, se detalla la implementación del mismo. Para realizar dicha implementación 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].

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 facilmente, y se pueden encontrar funciones equivalentes.

Interfaz gráfica

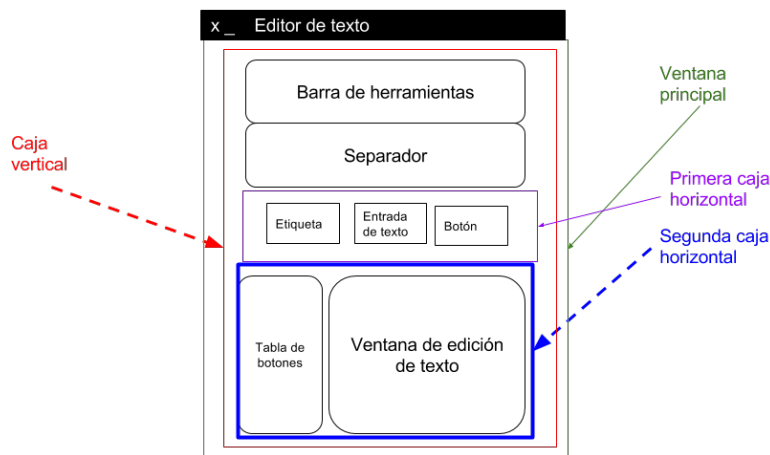


En la parte superior de la ventana de la aplicación se encuentra una barra de herramientas con los botones que proveen las funcionalidades. Los iconos de dichos botones son los que vienen por defecto en la librería GTK [7].

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).



Funcionalidades.

Funcionalidades básicas

Funcionalidades básicas de un editor de texto: Abrir archivo (con ventana de dialogo), guardar archivo, editar archivo, pegar lo que se tenga en el clipboard (equivalente a presionar CTRL+C) a la ventana de edición de texto, copiar el texto seleccionado al clipboard.

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 está 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 está 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.

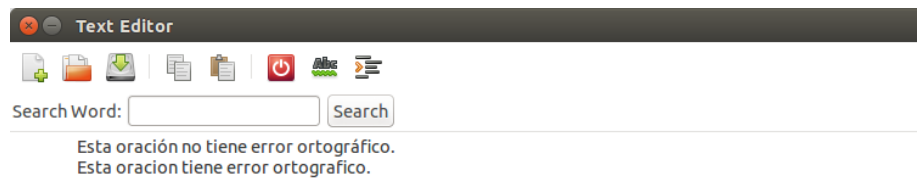
Por último, en cuando al resaltado de los comentarios, el mismo se realiza en caso de que haya sintaxis válida de Haskell. Para lograr este resaltado, se buscan ocurrencias en el texto de dos guiones seguidos (“--”); cuando esto ocurre, se marca como comentario a todo el texto que se encuentre desde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

Corrector ortográfico

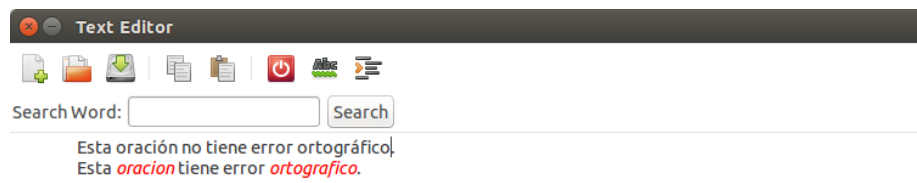
Para detectar las palabras mal escritas, se emplea la librería Aspell para Haskell (Hspell) [6]. Esta librería indica como incorrectas a aquellas palabras que no se encuentren en el diccionario que se está empleando; la implementación hecha para este trabajo ofrece soporte para el diccionario español.

En cuanto al código, 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 “spellCheckerWithOptions” de la librería Aspell para determinar si la palabra está correctamente escrita. En caso de que dicha palabra no sea correcta, se procede a marcarla en la ventana de edición de texto.

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. 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.

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.

En cuanto al código, lo que se realiza internamente es utilizar 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.

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.

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.

Copiar

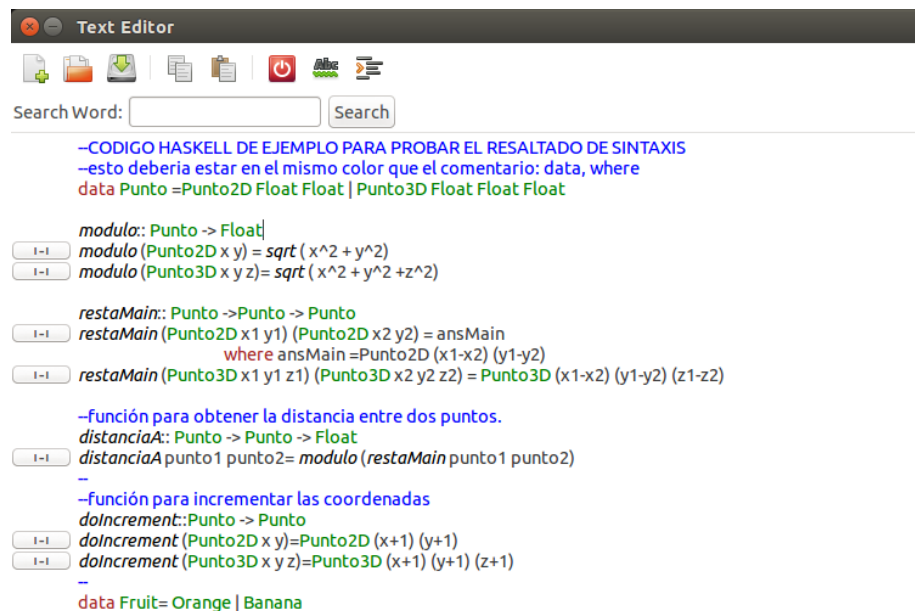
Se copia en el clipboard, lo que se haya seleccionado de la ventana principal de edición de texto. Para lograr esto, se obtiene el clipboard de selección de la interfaz gráfica y el clipboard general del sistema operativo. Por último, el texto seleccionado en el clipboard de la interfaz gráfica (texto marcado en la ventana de edición) se graba en el clipboard del sistema operativo.

Pegar

Se copia el contenido del clipboard en la posición del cursor en la ventana de edición de texto. Internamente, se obtiene el clipboard del sistema operativo, 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.

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:



The screenshot shows a window titled "Text Editor" with a toolbar containing icons for file operations and a search bar. The search bar contains the text "Search Word:" and a "Search" button. The code in the editor is as follows:

```
--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 (Punto2D x1 y1) (Punto2D x2 y2) = ansMain
[−]           where ansMain = Punto2D (x1-x2) (y1-y2)
[−] 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
```

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:

The screenshot shows a 'Text Editor' window with a toolbar and a search bar. The code is as follows:

```

--CODIGO HASKELL DE EJEMPLO PARA PROBAR EL RESALTADO DE SINTAXIS
--esto debería 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
  
```

Interactive buttons (labeled 'I-I') are placed next to function definitions: `modulo`, `restaMain`, `distanciaA`, and `doIncrement`. The `restaMain` button is currently active, indicated by a red border. The `data` keyword is highlighted in red, and the `Fruit` type is highlighted in green.

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.

En cuanto al código implementado, esta funcionalidad empieza 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”. En forma similar al proceso de marcado de sintaxis (ver análisis hecho en la sección “Resaltado de sintaxis de haskell”), se

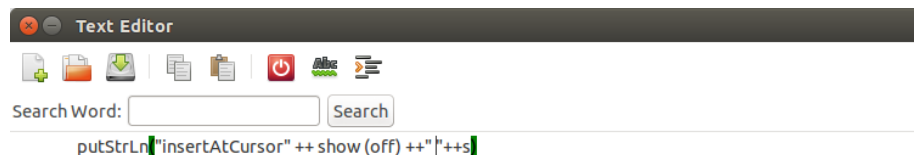
procesa los distintos valores contenidos hasta llegar a los valores tipo “HsMatch”. Cuando esto último ocurre, se considera al final del nombre de la función como posición inicial del texto a ocultar, y como posición final se toma el comienzo de la siguiente declaración de función; en caso de que no exista una siguiente definición de función, se considera como posición final a cualquier declaración de Haskell que se encuentre luego de la definición de la función actual.

Una vez que se tiene la posición inicial y final 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 le setea 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. 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.

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:

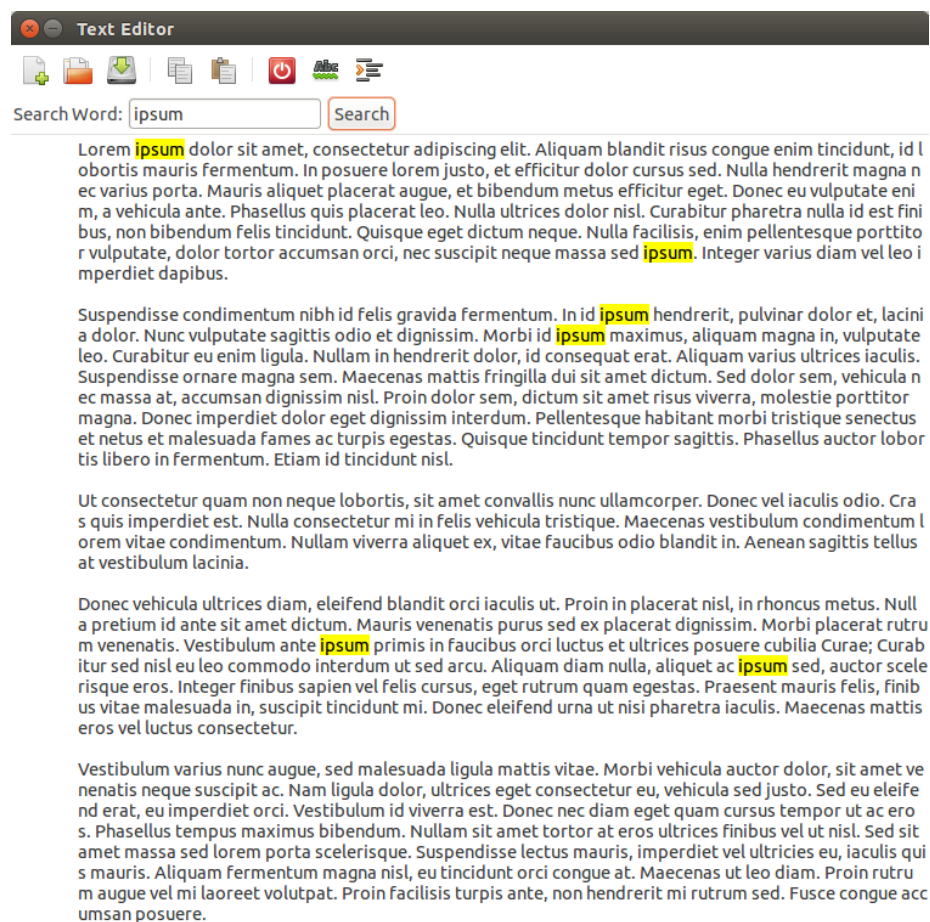


Debido a las limitaciones de la interfaz gráfica utilizada, no se pudo encontrar una forma de interceptar el evento luego de que se mueve el cursor en el texto, solo el evento antes de moverse. Por ese motivo, se resaltan los parentesis que contienen la posicion anterior luego de mover el cursor. Si 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.

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.



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.

Código Relevante

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

Función main ubicada en Main2.hs. Función principal donde se inicializa la interfaz gráfica.

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” .

Compilación y ejecución del proyecto

Antes de compilar el programa, se necesitan tener instaladas las dependencias descritas 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”.

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>