INSTITUTO TECNOLÓGICO DE MORELIA

División De Estudios Profesionales

Departamento De Sistemas Y Computación

PROGRAMACIÓN LÓGICA Y FUNCIONAL

Unidad II: Modelo de Programación Funcional.

PROYECTO CHATBOT: Aplicaciones de Listas y Árboles.

PROFESOR

ALUMNOS

Gil Jacome Arturo - 14121426
Martínez Aguilar Diego Ulises - 15121185

MORELIA, MICHOACÁN, MÉXICO
viernes, 17 de mayo de 2019

Tabla de contenido

CHAT BOT TAQUERO

DESARROLLO

Fase 0: Creación de las listas frases, símbolos, stopwords y consultas.

```haskell
import Data.List
import Data.Ord

frases = ["las carnitas son nutritivas","los tacos estan muy ricos", "ya no hay al pastor", "aun hay de bistec", "hoy hay tacos de todo", "cuestan baratos los tacos", "la salsa es la mas rica de la region", "aqui esta tu cuenta", "gracias a ti"]

simbolos = [',', '.', '!', '?', '¿', '¡', '#', '$', '%']

stopwords = ["para", "que", "me", "mi", "al", "su", "a", "el", "la", "los", "las", "un", "una", "unos","unas","es", "muy", "tiene", "se", "tengo", "a", "ante", "bajo", "cabe", "con", "contra", "de", "desde", "durante", "en", "entre", "hacia", "hasta", "mediante", "para", "por", "según", "sin", "sobre", "tras", "versus", "vía"]

consultas=["de que hay tacos hoy?","hay tacos al pastor?","tiene de bistec?","que tan baratos estan los tacos?","muchas gracias!!!","me das mi cuenta $$$"]
```

Figura 1: Se crean las listas correspondientes a las **frases**, **símbolos**, **stopwords** y **consultas**. El módulo List de Haskell tiene muchas funciones para ordenar, modificar y construir listas, para ello se importan las clases **Data.List** y **Data.Ord**.

Fase 1: Se limpia el texto de símbolos tipográficos.

```haskell
controlador = map calcula consultas

calcula texto = let
                    nuevoTexto = [ caracter | caracter <- texto, not(caracter `elem` simbolos)]
                in
                    nuevoTexto
```

Figura 2: Creamos la función **controlador** la cual hará un mapeo con la función **map** utilizada para aplicar la función **calcula** a todos los elementos de la lista **consultas**. **Let** nos permite crear una lista de declaraciones anidadas, de ámbito léxico y recursivas recíprocamente, dentro de ella buscamos en la lista los **símbolos** de cada una de las **consultas** eliminando los símbolos tipográficos haciendo uso de **elem** que es el predicado de pertenencia a la lista.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                    1: ghci

*Main> :load "chat final.hs"
[1 of 1] Compiling Main             ( chat final.hs, interpreted )
Ok, one module loaded.
*Main> consultas
["de que hay tacos hoy?","hay tacos al pastor?","tiene de bistec?","que tan baratos estan los tacos?","muchas gracias!!!","me das mi cuenta $$$"]
*Main> controlador
["de que hay tacos hoy","hay tacos al pastor","tiene de bistec","que tan baratos estan los tacos","muchas gracias","me das mi cuenta "]
*Main>
```

Figura 3: Hacemos uso de la terminal de nuestro editor de texto **Visual Studio Code** para ejecutar nuestro compilador **ghci**, cargamos el archivo usando el comando **:load "chat final.hs"**, consultamos la lista **consultas** y llamamos al **controlador** para verificar que la lista ha sido limpiada de de símbolos tipográficos.

Fase 2: se eliminan las stop-words (palabras que no dan significado).

```
12   controlador = map calcula consultas
13
14   calcula texto = let
15                       nuevoTexto = [ caracter | caracter <- texto, not(caracter `elem` simbolos)]
16                       listaTexto = borrarStopWords nuevoTexto
17                       listaFrases = map borrarStopWords frases
18                   in
19                       listaTexto
20
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                                    1: ghci
*Main> consultas
["de que hay tacos hoy?","hay tacos al pastor?","tiene de bistec?","que tan baratos estan los tacos?","muchas gracias!!!","me das mi cuenta $$$"]
*Main> controlador
["de que hay tacos hoy","hay tacos al pastor","tiene de bistec","que tan baratos estan los tacos","muchas gracias","me das mi cuenta "]
*Main> :r
[1 of 1] Compiling Main             ( chat final.hs, interpreted )
Ok, one module loaded.
*Main> controlador
[["hay","tacos","hoy"],["hay","tacos","pastor"],["bistec"],["tan","baratos","estan","tacos"],["muchas","gracias"],["das","cuenta"]]
*Main>
```

Figura 4: Utilizamos la función **borrarStopWord** anteriormente creada al nuevo texto, y lo denominamos **listaTexto**, recargamos en la terminal el archivo y volvemos a llamar al **controlador** y verificamos que cada una de las consultas se a convertido en una lista de palabras sin las stopwords.

```
32  borrarStopWords string = let
33          listWords = words string
34          newList = [ word | word <- listWords, not(word `elem` stopwords)]
35      in
36          newList
37
```

Figura 5: La función **borrarStopWord** recibe como parámetro el texto denominado **string**, se aplica la función **words** que rompe la cadena en una lista de palabras, que están delimitadas por espacios en blanco y lo guardamos como **listWords**. Luego se busca en la lista las **stopwords** de cada una de las **consultas** eliminando las stop-words haciendo uso de **not elem**.

Fase 3: Lematizar, agrupar las palabras en términos comunes.

Q1=”tacos”

Q2=”bistec”,”pastor”

Q3=”hay”,”están”,”todo”

Q4=”ricos”,”baratos”

Q5=”cuenta”,”gracias”

Fase 4: Crear el diccionario, agrupar en un conjunto de palabras las keywords.

```
12   controlador = map calcula consultas
13
14 calcula texto = let
15                     nuevoTexto = [ caracter | caracter <- texto, not(caracter `elem` simbolos)]
16                     listaTexto = borrarStopWords nuevoTexto
17                     listaFrases = map borrarStopWords frases
18                     nuevaListaFrases = map (eliminarRepeticiones listaTexto) listaFrases
19                     diccionario = unir ([listaTexto]++nuevaListaFrases)
20                 in
21                     diccionario
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                                  1: ghci

*Main> controlador
[["hay","tacos","hoy","carnitas","son","nutritivas","estan","ricos","ya","no","pastor","aun","bistec","todo","cuestan","baratos","salsa","mas","rica
","region","aqui","esta","tu","cuenta","gracias","ti"],["hay","tacos","pastor","carnitas","son","nutritivas","estan","ricos","ya","no","aun","bistec
","hoy","todo","cuestan","baratos","salsa","mas","rica","region","aqui","esta","tu","cuenta","gracias","ti"],["bistec","carnitas","son","nutritivas"
,"tacos","estan","ricos","ya","no","hay","pastor","aun","hay","hoy","hay","tacos","todo","cuestan","baratos","tacos","salsa","mas","rica","region","
aqui","esta","tu","cuenta","gracias","ti"],["tan","baratos","estan","tacos","carnitas","son","nutritivas","ricos","ya","no","hay","pastor","aun","ha
y","bistec","hoy","hay","todo","cuestan","salsa","mas","rica","region","aqui","esta","tu","cuenta","gracias","ti"],["muchas","gracias","carnitas","s
on","nutritivas","tacos","estan","ricos","ya","no","hay","pastor","aun","hay","bistec","hoy","hay","tacos","todo","cuestan","baratos","tacos","salsa
","mas","rica","region","aqui","esta","tu","cuenta","ti"],["das","cuenta","carnitas","son","nutritivas","tacos","estan","ricos","ya","no","hay","pas
tor","aun","hay","bistec","hoy","hay","tacos","todo","cuestan","baratos","tacos","salsa","mas","rica","region","aqui","esta","tu","gracias","ti"]]
*Main>
```

Figura 6: Para la creación del diccionario creamos una **nuevaListaFrases** a partir del mapeo de la función **eliminarRepeticiones** creada anteriormente, y posteriormente a ello creamos el diccionario a través de **unir**, función creada anteriormente. Recargamos el archivo y llamamos al **controlador** y verificamos que para cada una de frases se crea un conjunto de palabras denominado **diccionario** en una lista por cada una de las posibles **consultas**.

```
27
28   unir [] = []
29   unir (x:xs) = x ++ unir xs
30
31   eliminarRepeticiones [] [] = []
32   eliminarRepeticiones xs ys = [ y | y <- ys, not(y `elem` xs) ]
33
```

Figura 7: La función **unir** recibe como parámetro una lista la cual concatena cada uno de sus elementos de lista y la función **eliminarRepeticiones** recibe como parametro dos listas que asemejan o comparan entre sí a través del **not elem** eliminando las repeticiones de la palabra para el diccionario, sintaxis propia de listas intencionales.

Fase 5: Crear **vectores** de cada consulta/frase.

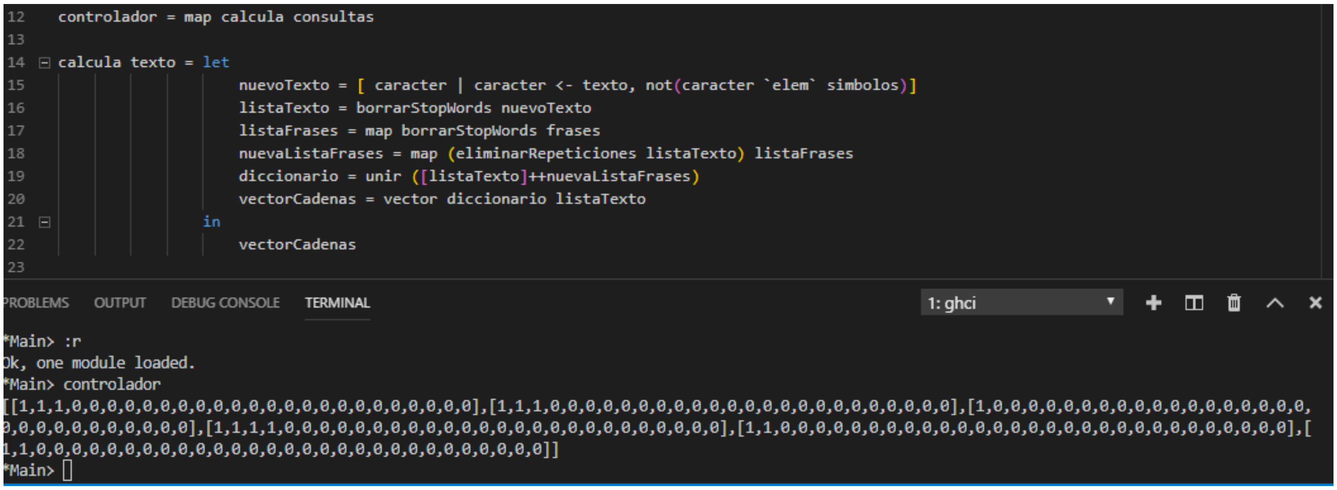

Figura 8: Creamos los vectores de cada una de las **consultas** una vez construida la **listaTexto** junto al ya creado diccionario a través de la función vector. Verificamos en la terminal recargando el archivo y llamando a la función **controlador** donde se aprecia una lista de vectores constituidos por 0's y 1's.

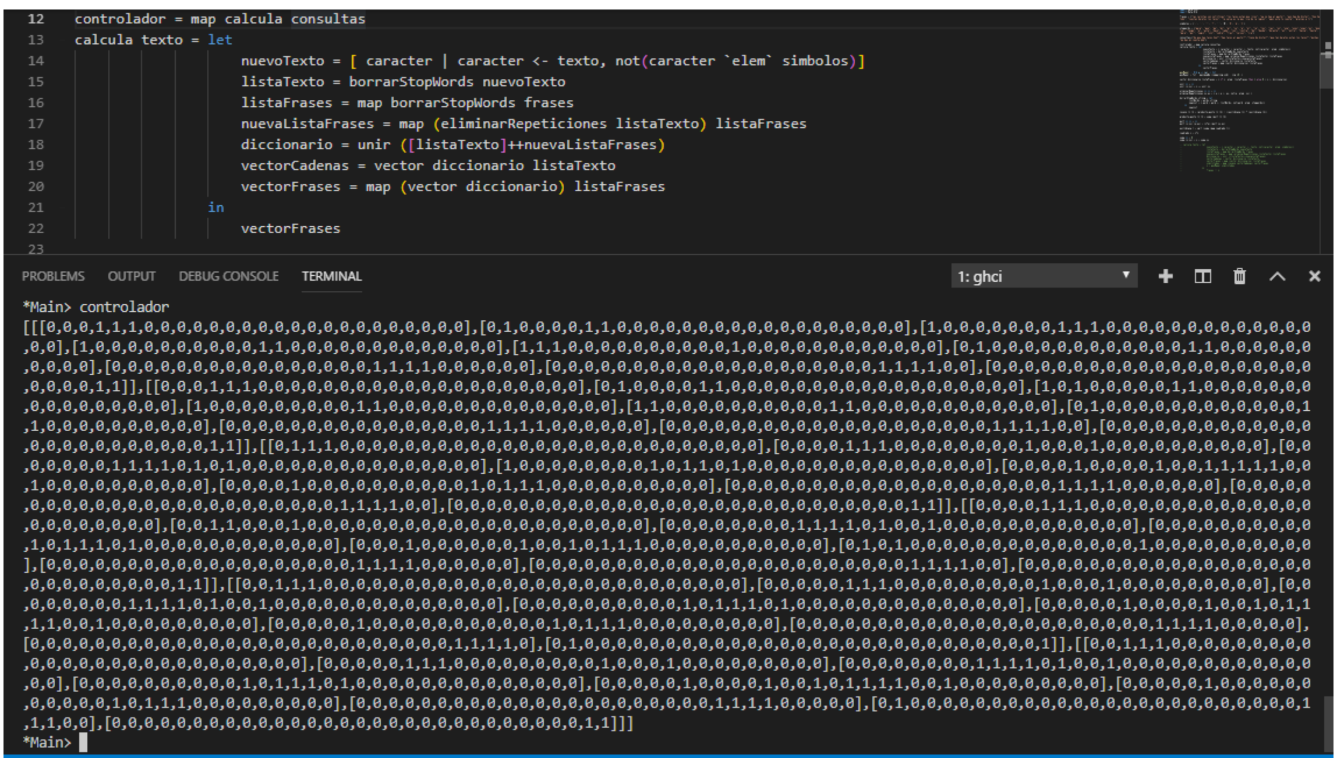

Figura 9: Ahora bien, se calculan los vectores para cada **frases** una vez construida la **listaFrases** respecto al ya creado diccionario a través del mapeo de la función vector. Verificamos en la terminal recargando el archivo y llamando a la función **controlador** donde se aprecia una lista de vectores constituidos por 0's y 1's. El número de elementos de cada vector indica la cantidad de palabras de cada frase.

```
26
27   vector diccionario listaFrases = [ if x `elem` listaFrases then 1 else 0 | x <- diccionario]
28
```

Figura 10: La función **vector** nos permite crear una lista con valores de 0 o 1, recibe como parámetros el diccionario de la y la lista de frases. Se tienen un *if* que condiciona que para cada elemento x de la lista **listaFrases**, debe devolverse la aplicación de la función denominada generador *x<-diccionario*, mientras que la parte del código *if x ´elem´ listaFrases then 1 else 0* que es la expresión que se usa para cada elemento generado en la parte derecha, entonces reemplaza el elemento x por 1 si la palabra está de lo contrario coloca un 0.

Fase 6: Cálculo de similitud.

```
12   controlador = map calcula consultas
13
14   calcula texto = let
15                       nuevoTexto = [ caracter | caracter <- texto, not(caracter `elem` simbolos)]
16                       listaTexto = borrarStopWords nuevoTexto
17                       listaFrases = map borrarStopWords frases
18                       nuevaListaFrases = map (eliminarRepeticiones listaTexto) listaFrases
19                       diccionario = unir ([listaTexto]++nuevaListaFrases)
20                       vectorCadenas = vector diccionario listaTexto
21                       vectorFrases = map (vector diccionario) listaFrases
22                       similitudes = map (coseno vectorCadenas) vectorFrases
23                   in
24                       similitudes
25
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                                   1: ghci

*Main> controlador
[[0.0,0.33333333333333337,0.2886751345948129,0.33333333333333337,0.8660254037844387,0.33333333333333337,0.0,0.0,0.0],[0.0,0.33333333333333337,0.57735026918962
58,0.33333333333333337,0.5773502691896258,0.33333333333333337,0.0,0.0,0.0],[0.0,0.0,0.0,0.4472135954999579,0.0,0.0,0.0,0.0,0.0],[0.0,0.5773502691896258,0.0,0.
0,0.20412414523193154,0.5773502691896258,0.0,0.0,0.0],[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.4999999999999999],[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.3535533905932373,0.
0]]
*Main>
```

Figura 11: Para calcular las similitudes de cada una de las **frases** dado el **vectorFrases,** para cada una de las posibles **consultas** dado el **vectorCadenas,** mapeamos la función coseno previamente creada. Para verificar recargamos el archivo y llamamos la función **controlador** que nos muestra la lista de listas de cada vector para cada consulta.

```
44   coseno l1 l2 = (producto_punto l1 l2) / ((euclidiana l1) * (euclidiana l2))
45
46   producto_punto l1 l2 = suma (mult l1 l2)
47
48   mult [] [] = []
49   mult (x:xs) (y:ys) = (x*y):(mult xs ys)
50
51   euclidiana l = sqrt (suma (map cuadrado l))
52
53   cuadrado x = x*x
54
55   suma [] = 0
56   suma (x:xs) = x + suma xs
```

Figura 12: La función coseno recibe como parámetro dos listas, el **vectorCadenas** y el **vectorFrases**, para ello se utilizan las siguiente fórmula: $coseno(c, f_1) = \frac{v(c)\cdot v(f_1)}{|v(c)|\cdot|v(f_1)|} = \frac{\sum_{i=1}^{n}(v(c)_i * v(f_1)_i)}{\sqrt{\sum_{i=1}^{n}(v(c)_i{}^2)} * \sqrt{\sum_{i=1}^{n}(v(f_1)_i{}^2)}}$

Fase 7: Selección de la frase con mayor similitud dada una consulta.

```
14  calcula texto = let
15                      nuevoTexto = [ caracter | caracter <- texto, not(caracter `elem` simbolos)]
16                      listaTexto = borrarStopWords nuevoTexto
17                      listaFrases = map borrarStopWords frases
18                      nuevaListaFrases = map (eliminarRepeticiones listaTexto) listaFrases
19                      diccionario = unir ([listaTexto]++nuevaListaFrases)
20                      vectorCadenas = vector diccionario listaTexto
21                      vectorFrases = map (vector diccionario) listaFrases
22                      similitudes = map (coseno vectorCadenas) vectorFrases
23                      n = posMayor similitudes
24                  in
25                      frases !! n
26
27  posMayor :: Ord a => [a] -> Int
28  posMayor = fst . maximumBy (comparing snd) . zip [0..]
29
```

Figura 13: Una vez obtenido la el vector de **similitudes** se determina la posición del elemento con el valor de similitud mayor, la cual hace referencia a la frase resultante, para ello usamos la función **posMayor** donde recibe una lista y retorna un valor entero, utiliza la función *maximumBy* que devuelve el valor máximo de una lista, que debe ser no vacío, finito y de un tipo ordenado. Además, suministramos la función de comparación *comparing*. Finalmente sacamos el índice n de la lista de frases.

RESULTADOS

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

*Main> calcula "de que hay tacos hoy?"
"hoy hay tacos de todo"
*Main> calcula "tiene de bistec?"
"aun hay de bistec"
*Main> calcula "y los de pastor?"
"ya no hay al pastor"
*Main> calcula "¿que tan baratos estan los tacos?"
"cuestan baratos los tacos"
*Main> calcula "me das mis cuenta $$$?"
"aqui esta tu cuenta"
*Main> calcula "muchas gracias!"
"gracias a ti"
*Main>
```

Figura 14: Probamos el chatbot taquero como se muestra en la imagen, con lo cual verificamos que realmente funciona.

ARBOLES

Arboles.hs | chat final.hs

```haskell
data Tree = Branch Int Tree Tree | Leaf Int

arbol1 = Branch 11 (Branch 7 (Leaf 5) (Leaf 9)) (Leaf 6)
arbol2 = Branch 1 (Branch 8 (Leaf 7) (Leaf 9)) (Branch 4 (Leaf 5) (Branch 4 (Leaf 5) (Branch 4 (Leaf 5) (Leaf 2))))
arbol3 = Branch 1 (Branch 8 (Leaf 6) (Branch 4 (Leaf 2) (Leaf 9))) (Branch 4 (Leaf 5) (Leaf 2))

suma (Leaf x) = x
suma (Branch x izq der) = x + (suma izq) + (suma der)

contar (Leaf x) = 1
contar (Branch x izq der) = 1 + (contar izq) + (contar der)

construirLista (Leaf x) = [x]
construirLista (Branch x izq der) = (construirLista izq)++[x]++(construirLista der)

profundidad (Leaf x) = 1
profundidad (Branch x izq der) = let
    profundidadizq=profundidad izq
    profundidadder=profundidad der
    in  if (profundidadizq>profundidadder) then 1 + profundidadizq else 1 + profundidadder
```

Figura 15: Creamos la estructura de un árbol binario, y creamos 3 árboles de ejemplo, posterior a ello desarrollamos algunas funciones básicas para el manejo de árboles.

EJERCICIOS

1. Indicar la suma de la rama mayor.

```haskell
--1. Indicar la suma de la rama mayor.
sumaProf (Leaf x) = 1
sumaProf (Branch x izq der) = let
    sumaProfIzq=sumaProf izq
    sumaProfDer=sumaProf der
    in  if (sumaProfIzq>sumaProfDer) then suma izq else suma der
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: ghci

```
"hoy hay tacos de todo"
*Main> :load "Arboles.hs"
[1 of 1] Compiling Main             ( Arboles.hs, interpreted )
Ok, one module loaded.
*Main> sumaProf arbol1
21
*Main> sumaProf arbol3
29
*Main>
```

Figura 16: Cargamos el archivo *Arboles.hs* y realizamos un par de pruebas con los árboles que creamos anteriormente.

2. Dada una lista de de árboles sumar los valores de todos los nodos de todos los árboles.

```
29   --2. Dada una lista de de arboles sumar los valores de todos los nodos de todos los arboles.
30   concatenarListas [] = []
31   concatenarListas (x:xs) = x ++ concatenarListas xs
32
33   sumaTodo [] = 0
34   sumaTodo (x:xs) = x + sumaTodo xs
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                    1: ghci
*Main> calcula "de que hay tacos hoy?"
"hoy hay tacos de todo"
*Main> :load "Arboles.hs"
*Main> sumaTodo (concatenarListas (map construirLista [arbol1, arbol2, arbol3]))
133
*Main>
```

Figura 17: Se ingresa la estructura siguiente: *sumaTodo (concatenarListas (map construirLista [arbol1, arbol2, arbol3])).*

3. Dada una lista de árboles, indicar el número mayor de nodos de ellos.

```
37   --3. Dada una lista de arboles, indicar el número mayor de nodos de ellos
38   mayor [] v = v
39   mayor (x:xs) v = if(x>v) then mayor xs x else mayor xs v
40
41   elmayor [] = -1
42   elmayor (x:xs) = mayor xs x
43       --elmayor (map length (map construirLista [arbol1, arbol2, arbol3]))
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                    1: ghci
*Main> elmayor (map length (map construirLista [arbol1, arbol2, arbol3]))
11
*Main>
```

Figura 18: Se ingresa la estructura siguiente: *elmayor (map length (map construirLista [arbol1, arbol2, arbol3])).*