Lógica Computacional
Práctica 01: Introducción a Haskell

Erik Rangel Limón Manuel Soto Romero

Semestre 2024-2
Facultad de Ciencias UNAM

Objetivos

1. Familiarizar a los alumnos con el lenguaje de programación *Haskell*, y poder abordar problemas con programación funcional.
2. Trabajar con listas, listas por comprensión y con definiciones inductivas de nuevos tipos de dato.

Instrucciones

1. Para esta práctica necesitan tener instalado *GHC* y *Cabal*.

   Pueden usar una de las siguientes dos alternativas.

   - **ghcup:**
     Esta es una opción recomendada para instalar las dos herramientas usando un mismo instalador.
     Las instrucciones de cómo instalarlo para sistemas basados en *Unix* y *Windows* vienen en la página `https://www.haskell.org/ghcup/`.
     Una vez instalado, pueden verificar la instalación de `ghc` y de `cabal` utilizando el comando `ghcup tui`, el cual abrirá una interfaz de terminal (utilicen las versiones recomendadas de `ghc` y de `cabal`).
   - **Administrador de paquetes:**
     Por lo general los paquetes que necesitan llevan los nombres `ghc` y `cabal-install`, investiguen si esto difiere según su distribución.
     Debian:

     ```
     sudo apt install ghc cabal-install
     ```

     Fedora:

     ```
     sudo dnf install ghc cabal-install
     ```

2. Necesitan también la biblioteca *QuickCheck*.

   Una vez instalados *GHC* y `Cabal`, lo pueden instalar con la siguientes líneas de comandos:

   ```
   cabal update
   cabal install --lib QuickCheck
   ```

   Pueden ver si la herramienta se instaló bien si pueden interpretar con `ghci` el archivo `src/Test.hs` y no salen errores.

   En la carpeta de la práctica:

```
ghci Test.hs
```

Pueden salir de `ghci` con `:q` o con la combinación de teclas `Ctrl+D`.

3. Resolver todas las funciones que se encuentran en el archivo `src/Intro.hs`.
4. Pueden verificar sus funciones y recibir una calificación **tentativa** con el archivo `src/Test.hs`.
   En la carpeta de la práctica:

```
ghci Test.hs
```

```
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Intro            ( Intro.hs, interpreted )
[2 of 2] Compiling Test             ( Test.hs, interpreted )
Ok, two modules loaded.
ghci> main
Pruebas traduceF:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
+++ OK, passed 1 test.
Pruebas palindromo:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
Pruebas prefijoComun:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
Pruebas mezcla:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
Pruebas diferencia:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
Pruebas productoPunto:
+++ OK, passed 1000 tests.
Pruebas triada:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
Pruebas combinaciones:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests; 40 discarded.
Pruebas binHfold:
+++ OK, passed 1000 tests.
Pruebas binHenum:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
Pruebas binFold:
+++ OK, passed 1000 tests.
Pruebas binEnum:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
Pruebas exitosas: 22/22
Calificación tentativa: 10.0
```

Ejercicios

1. *Traductor a "efe"*

   Dada una cadena de texto, regresa su traducción a lenguaje *"efe"*. A cada vocal de una palabra le sigue una 'f' y se repite la vocal. (Puedes asumir que las palabras no llevarán acentos).

```
traduceF :: String -> String
```

   Ejemplos:

```
ghci> traduceF "Hola, Mundo!"
"Hofolafa, Mufundofo!"
ghci> traduceF "Adios, Mundo!"
"Afadifiofos, Mufundofo!"
```

2. *Mezcla*

   Dadas dos listas ordenadas, las mezcla preservando el orden. Debes asumir que las listas recibidas como parámetro ya están ordenadas, y por tanto la complejidad de la función debe ser $O(n+m)$, siendo n y m la longitud de las listas respectivamente.

```
mezcla :: [Int] -> [Int] -> [Int]
```

   Ejemplos:

```
ghci> mezcla [1,5,9] [2,8]
[1,2,5,8,9]
ghci> mezcla [52,79,100] [30,48,53,117,118]
[30,48,52,53,79,100,117,118]
```

3. *Palíndromo*

   Dada una cadena de texto, determina si ésta es un palíndromo. La cadena puede contener espacios en blanco y no deben ser tomados en cuenta, tampoco distingue entre mayúsculas o minúsculas.

```
palindromo :: String -> Bool
```

   Ejemplos:

```
ghci> palindromo "Somos o no somos"
True
ghci> palindromo "Acaso hubo buhos aca"
True
ghci> palindromo "Las rosas son rojas"
False
```

4. *Prefijo común*

   Dada una lista de cadenas devuelve el prefijo que tienen en común todas las cadenas de la lista.

```
prefijoComun :: [String] -> String
```

   Ejemplo:

```
ghci> prefijoComun ["prefijo","preludio","prenda"]
"pre"
ghci> prefijoComun ["funcional","funcionamiento","fundamental"]
"fun"
ghci> prefijoComun ["facultad","involucrar","caminar"]
""
```

5. *Diferencia*

   Dadas dos listas de elementos comparables, regresa la lista resultado de eliminar de la primera lista los elementos que aparezcan en la segunda.

   Puedes asumir que las listas recibidas como parámetro no tienen elementos repetidos.

```
diferencia :: (Eq a) => [a]-> [a] -> [a]
```

```
ghci> diferencia  [1,3,5,2,10,20] [3,18,2,21,10,50]
[1,5,20]
ghci> diferencia [1,3..11] [2,4..10]
[1,3,5,7,9,11]
```

6. *Producto punto*

   Dadas dos listas de enteros, calcula el producto punto de ambas listas.

   Utiliza listas por comprensión

```
productoPunto :: [Integer] -> [Integer] -> Integer
```

   Ejemplos:

$$\begin{aligned}
\texttt{productoPunto [5,3,2] [2,4]} &= 5\cdot 2+5\cdot 4+3\cdot 2+3\cdot 4+2\cdot 2+2\cdot 4\\
&= 60\\
\texttt{productoPunto [1,4,5,2,3] [10,32,21]} &= 1\cdot 10+1\cdot 32+1\cdot 21+4\cdot 10+4\cdot 32+4\cdot 21+5\cdot 10+5\cdot 32\\
&\quad +5\cdot 21+2\cdot 10+2\cdot 32+2\cdot 21+3\cdot 10+3\cdot 32+3\cdot 21\\
&= 945
\end{aligned}$$

7. *Triadas pitagóricas*

   Dado un número natural n, regresa todas las combinaciones de números naturales (a,b,c) tales que $a < b < c \leq n$ y $a^2 + b^2 = c^2$.

```
triada :: Int -> [(Int,Int,Int)]
```

```
ghci> triada 10
[(3,4,5),(6,8,10)]
ghci> triada 15
[(3,4,5),(5,12,13),(6,8,10),(9,12,15)]
ghci> triada 20
[(3,4,5),(5,12,13),(6,8,10),(8,15,17),(9,12,15),(12,16,20)]
```

8. *Combinaciones*

Dada una lista de elementos comparables, regresa todos los subconjuntos posibles de tres elementos. Debes asumir que todos los elementos de la lista recibida como parámetro son distintos.

No debe haber elementos repetidos, considera que dos triplas son iguales si tienen los mismos elementos en cualquier orden, es decir:

$$(a,b,c) = (b,c,a)$$

```
combinaciones :: (Eq a) => [a] -> [(a,a,a)]
```

Ejemplos:

```
ghci> combinaciones ["Marta","Maria","Juan","Ximena","Diego"]
[("Marta","Maria","Juan"),("Marta","Maria","Ximena"),("Marta","Maria","Diego")
,("Marta","Juan","Ximena"),("Marta","Juan","Diego"),("Marta","Ximena","Diego")
,("Maria","Juan","Ximena"),("Maria","Juan","Diego"),("Maria","Ximena","Diego")
,("Juan","Ximena","Diego")]
ghci> combinaciones "abcdef"
[('a','b','c'),('a','b','d'),('a','b','e'),('a','b','f'),('a','c','d')
,('a','c','e'),('a','c','f'),('a','d','e'),('a','d','f'),('a','e','f')
,('b','c','d'),('b','c','e'),('b','c','f'),('b','d','e'),('b','d','f')
,('b','e','f'),('c','d','e'),('c','d','f'),('c','e','f'),('d','e','f')]
```

9. *Doblar un árbol*

Para éste y el siguiente ejercicio considera la siguiente definición de árbol binario con sólo elementos en sus hojas

```
data BinH a = Hoja a
            | Rama (BinH a) (BinH a) deriving Show
```

Dada una función f, un valor "final" y un árbol binario, debes regresar el resultado de aplicar la funcion f a lo largo de todo el árbol usando como caso base el valor final.

Debe aplicarse en orden, es decir, desde la hoja más a la izquierda, hasta la que está más a la derecha.

```
binHfold :: (a -> b -> b) -> b -> BinH a -> b
```

Supongamos que `binHtree` es el siguiente árbol

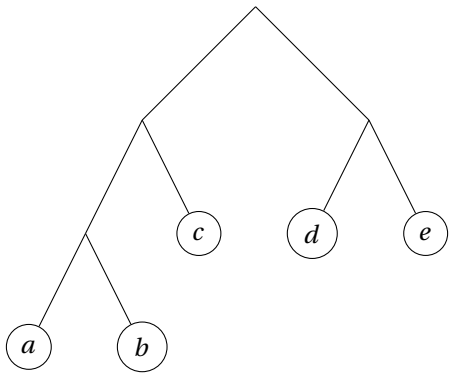

Figura 1: `binHtree`

En Haskell, éste árbol sería representado de la siguiente forma:

```
binHtree = Rama (Rama (Rama (Hoja a) (Hoja b)) (Hoja c)) (Rama (Hoja d) (Hoja e))
```

La aplicación de binHFold debería verse de la siguiente forma:

```
binHfold fun final binHtree = fun a (fun b (fun c (fun d (fun e final))))
```

Por ejemplo:

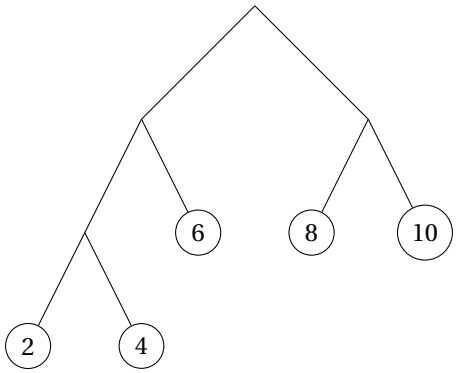

Figura 2: `numHtree`

```
binHfold (+) 0 numHtree = 2 + (4 + (6 + (8 + (10 + 0)))) = 30
```

10. *Enumerar un árbol*

Dado un árbol binario, enumera las hojas de izquierda a derecha (empezando por el cero).

```
binHenum :: BinH a -> BinH (Int,a)
```

Consideremos al árbol `binHtree`:

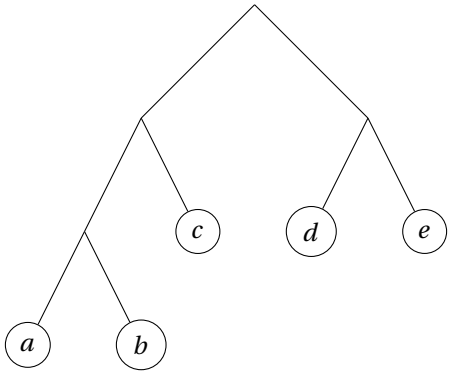

Figura 3: `binHtree`

El resultado de la aplicación `binHenum binHtree` debe verse como sigue:

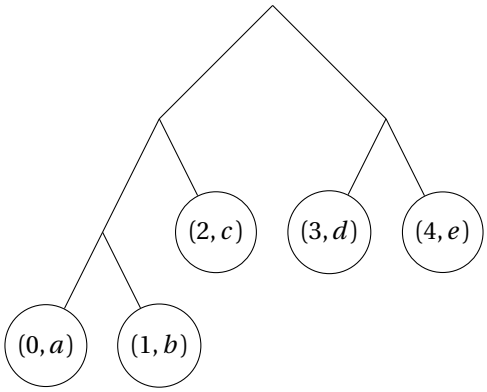

Figura 4: `binHenum binHtree`

11. *Doblar un árbol: Segunda parte*

Para éste y el siguiente ejercicio considera la siguiente definición de árbol binario con elementos en sus nodos

```
data Bin a = Vacio
           | Nodo a (Bin a) (Bin a) deriving Show
```

Similar a `binHFold`, pero el orden en el que se debe aplicarse es "inorder".

```
binFold :: (a -> b -> b) -> b -> Bin a -> b
```

Supongamos que `binTree` es el siguiente árbol:

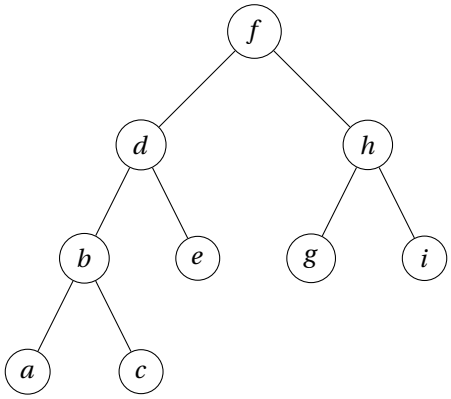

Figura 5: `binTree`

En Haskell éste árbol sería representado de la siguiente forma:

```
binTree = Nodo f (Nodo d (Nodo b (Nodo a Vacio Vacio)
                                 (Nodo c Vacio Vacio))
                         (Nodo e Vacio Vacio))
                 (Nodo h (Nodo g vacio)
                         (Nodo i Vacio))
```

La aplicación de `binFold` debería verse de la siguiente forma:

```
binFold fun final binTree = fun a
                              (fun b
                                (fun c
                                  (fun d
                                    (fun e
                                      (fun f
                                        (fun g
                                          (fun h
                                            (fun i final)))))))))
```

Por ejemplo:

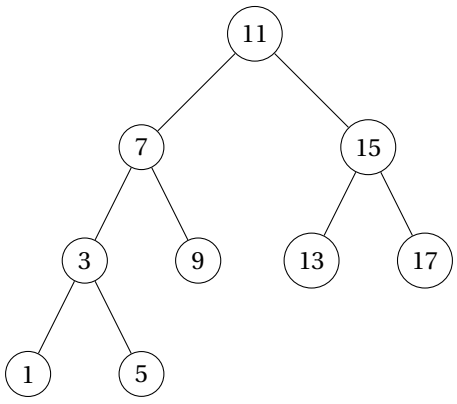

Figura 6: `numTree`

```
binFold (+) 0 numTree = 1 + (3 + (5 + (7 + (9 + (11 + (13 + (15 + (17 + 0)))))))) = 81
```

12. *Enumerar un árbol: Segunda Parte*

    Similar a `binHenum`, pero enumerando los nodos "inorder".

```
binEnum :: Bin a -> Bin (Int,a)
```

Ejemplo:

Consideremos el árbol `binTree`:

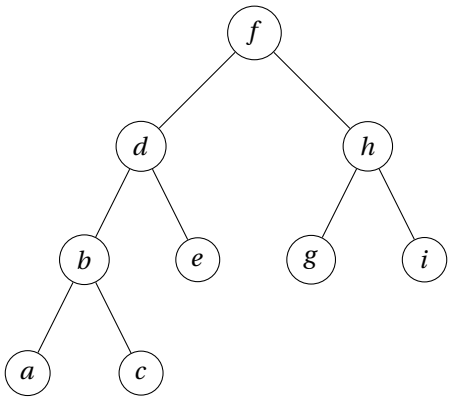

Figura 7: `binTree`

La aplicación de `binEnum binTree` debe regresar:

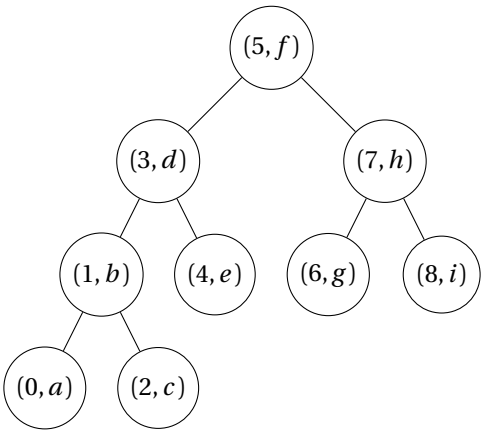

Figura 8: `binEnum binTree`

Entrega

1. La tarea se entrega en *Github Classroom*.
2. La tarea se entrega en equipos de hasta 4 personas.
3. Además de resolver los ejercicios, deben entregar un `README` (`.md` o `.org`) en la raiz de la carpeta de la práctica que contenga los datos de todos los integrantes, y por cada una de sus funciones dejar una breve explicación de su solución.

Consideraciones

1. Todas las prácticas con copias totales o parciales tanto en el código como en el `README` serán evaluadas con cero.
2. Las únicas funciones con soluciones iguales admisibles son todas aquellas que sean iguales a las resueltas por el grupo en el laboratorio, sin embargo la explicación de su solución en el `README` debe ser única para cada equipo.
3. No entregar el `README` o tenerlo incompleto se penalizará con hasta dos puntos menos sobre su calificación en la práctica.
4. Cada día de retraso se penalizará con un punto sobre la calificación de la práctica.
5. Pueden usar las funciones que provee la biblioteca `Data.Char` (`https://hackage.haskell.org/package/base-4.19.0.0/docs/Data-Char.html`) y las funciones `map, filter, reverse, sum, elem, all, any, head, tail`. Pueden utilizar otras funciones siempre y cuando no resuelvan directamente el ejercicio, expliquen en el `README` qué es lo que hace, y pertenezca a alguna biblioteca estándar de *Haskell* (es decir, no es válido utilizar paqueterías que requieran una instalación manual).
6. Los tests de ésta práctica tardan entre 30 segundos y 2 minutos. Al final del archivo `src/Intro.hs` hay una variable `pruebas`, ésta indica el número de pruebas que se realizarán a cada una de las funciones. Por defecto está configurada para hacer 1000 pruebas, pero pueden ponerle un valor menor si consideran que las pruebas tardan mucho en completarse, siempre y cuando éste valor no sea menor a 100. Si las pruebas tardan más de los dos minutos aún con 100 pruebas, consideren otra solución.
7. Si a la variable `pruebas` le pusieron un valor menor a 1000, asegúrense que pasen las pruebas de manera consistente, es decir, que puedan ejecutar varias veces las pruebas sin dar errores.
8. Pueden hacer tantas funciones auxiliares como quieran, pero no deben modificar la firma de las funciones ni de las variables, ni la definición de tipos de dato que se les dió.
9. No se recibirán prácticas que no compilen (no debe arrojar errores la orden `ghci Test.hs`). Si no resuelven alguna de las funciones déjenlas como `undefined`, pero no eliminen la función, ya que ésto lanzará errores.
10. No deben modificar el archivo `src/Test.hs`. Si encuentran errores o tienen dudas sobre las pruebas, manden un correo al ayudante de laboratorio (si encuentran errores tendrán una participación).
11. Las participaciones en el laboratorio se aplicarán de manera individual sobre la calificación que tuvieron en la práctica.