Lógica Computacional
Práctica 01: Introducción a Haskell

Erick Daniel Arroyo Martínez Erik Rangel Limón Manuel Soto Romero

Semestre 2025-2
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 9.8.2: https://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Intro            ( Intro.hs, interpreted )
[2 of 2] Compiling TestSolucion     ( Test.hs, interpreted )
Ok, two modules loaded.
ghci> main
Pruebas firstVowels:
+++ OK, passed 1000 tests.

Pruebas isAnagram:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.

Pruebas commonSuffix:
+++ OK, passed 1000 tests.

Pruebas interseccion:
+++ OK, passed 1000 tests.

Pruebas ackerman:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests; 4608 discarded.

Pruebas quicksort:
+++ OK, passed 1000 tests.

Pruebas bTreeSearch:
+++ OK, passed 1000 tests.

Pruebas bTreeInsert (bTreeSearch tras inserción):
+++ OK, passed 1000 tests.

Pruebas bTreeMap:
+++ OK, passed 1000 tests.

Pruebas bTreeHeight:
+++ OK, passed 1000 tests.
+++ OK, passed 1000 tests; 155 discarded.
+++ OK, passed 1000 tests.

Pruebas exitosas: 16/16
Calificación tentativa: 10.0
```

Ejercicios

1. *Primeras Vocales*
   Dada una cadena de texto, la función regresa una nueva cadena que contiene al inicio las vocales de la cadena original, preservando el orden, seguida de las consonantes.

```
firstVowels :: String -> String
```

   Ejemplos:

```
ghci> firstVowels "Hola, Mundo!"
"oauoHl, Mnd!"
ghci> firstVowels "Adios, Mundo!"
"Aiouods, Mnd!"
```

2. *Anagrama*
   Dadas dos cadenas de texto, la función determina si son anagramas, es decir, si contienen exactamente los mismos caracteres sin importar el orden. Una pista es ordenar ambas cadenas y luego compararlas.

```
isAnagram :: String -> String -> Bool
```

   Ejemplos:

```
ghci> isAnagram "roma" "amor"
True
ghci> isAnagram "Altisonancia" "Nacionalista"
True
```

3. *Sufijo Común*
   Dada una lista de cadenas, la función regresa el mayor sufijo común a todas ellas. Una estrategia es invertir cada cadena, buscar el prefijo común en las versiones invertidas y luego invertir el resultado.
   Debes usar foldr o foldl, en caso de no usarse deberá ser justificado, en caso contrario se dividirá la calificación obtenida en esta función entre dos.

```
commonSuffix :: [String] -> String
```

   Ejemplo:

```
ghci> commonSuffix ["caminando", "volando", "jugando"]
"ando"
```

4. *Intersección de Listas*
   Dadas dos listas, la función regresa una nueva lista con los elementos que se encuentran en ambas. Puedes utilizar comprensiones de listas o recursión para filtrar dichos elementos.

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

   Ejemplo:

```
ghci> intersection [1,2,3,4] [3,4,5,6]
[3,4]
```

5. *Función de Ackerman*
La función de Ackermann es una función matemática definida recursivamente que crece a un ritmo extraordinariamente rápido. Se utiliza en la teoría de la computación para demostrar ejemplos de funciones totales que no son primitiva recursivas. Esto significa que, a diferencia de muchas funciones definidas por recursión (como la suma, el producto o la exponenciación), la función de Ackermann no puede ser expresada mediante bucles con un número fijo de iteraciones.
Definición de la Función de Ackermann
La función de Ackermann, denotada por $A(m,n)$, se define para números naturales m y n con las siguientes reglas:

- **Caso base:** Si $m = 0$, entonces
$$A(0,n) = n+1.$$
- **Primera regla recursiva:** Si $m > 0$ y $n = 0$, entonces
$$A(m,0) = A(m-1,1).$$
- **Segunda regla recursiva:** Si $m > 0$ y $n > 0$, entonces
$$A(m,n) = A(m-1,A(m,n-1)).$$

Puedes consultar más sobre la *función Ackerman* en: `https://resources.saylor.org/wwwresources/archived/site/wp-content/uploads/2011/06/Ackermann-Function.pdf`

```haskell
ackerman :: Integer -> Integer -> Integer
```

Ejemplo:

```
ghci> ackerman 3 2
29
```

6. *Quicksort*
Implementa el algoritmo de ordenamiento rápido (quicksort) de manera recursiva. Una sugerencia es elegir un elemento como pivote y dividir la lista en dos partes: elementos menores y elementos mayores al pivote.

```haskell
quicksort :: (Ord a) => [a] -> [a]
```

Ejemplo:

```
ghci> quicksort [3,1,4,1,5,9,2,6]
[1,1,2,3,4,5,6,9]
```

7. *Árbol Binario de Búsqueda (BST)*
Se define la estructura de un árbol binario de búsqueda (BST) y se implementan funciones para insertar, buscar, aplicar una función a cada nodo y calcular la altura del árbol.

```haskell
data BTree a = Empty | Node a (BTree a) (BTree a) deriving (Show, Eq)
```

8. *Inserción en BST*
Inserta un elemento en un árbol binario de búsqueda, manteniendo la propiedad de orden. La inserción se realiza de forma recursiva comparando el elemento con el nodo actual.

```haskell
bTreeInsert :: (Ord a) => a -> BTree a -> BTree a
```

Ejemplo:

```
ghci> let tree = bTreeInsert 5 Empty
ghci> let tree' = bTreeInsert 3 tree
ghci> tree' Node 5 (Node 3 Empty Empty) Empty
```

9. *Búsqueda en BST*
   Busca un elemento en un árbol binario de búsqueda. Retorna `True` si el elemento se encuentra en el árbol y `False` en caso contrario, aprovechando la estructura ordenada del árbol para optimizar la búsqueda.

```
bTreeSearch :: (Ord a) => a -> BTree a -> Bool
```

Ejemplo usando *numHTree1*:

```
ghci> bTreeSearch 3 numHTree1
True
ghci> bTreeSearch 11 numHTree1
False
```

10. *Mapeo en BST*
    Aplica una función a cada elemento del árbol, retornando un nuevo BST con la misma estructura pero con los elementos transformados. Es un buen ejercicio para practicar la recursión en estructuras de datos.

```
bTreeMap :: (a -> b) -> BTree a -> BTree b
```

Ejemplo usando *numHTree1*:

```
ghci> bTreeMap (*2) numHTree1
Node 12
(Node 8 (Node 4 (Node 2 Empty Empty) (Node 6 Empty Empty)) (Node 10 Empty Empty))
(Node 16 (Node 14 Empty Empty) (Node 18 Empty (Node 20 Empty Empty)))
```

Noción Gráfica

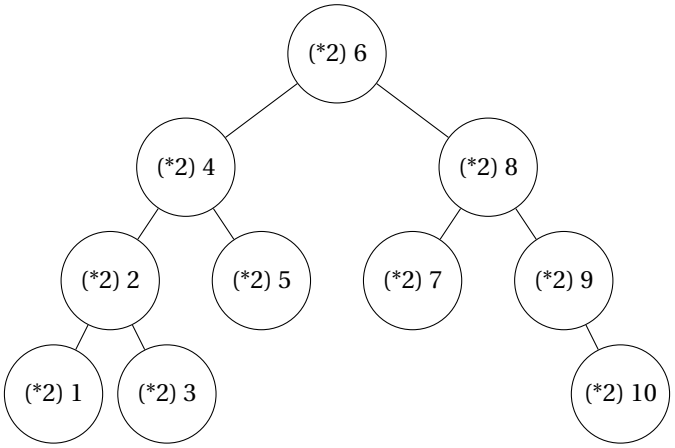

Figura 1: `(*2) aplicado a numHtree1`

Árbol binario resultante

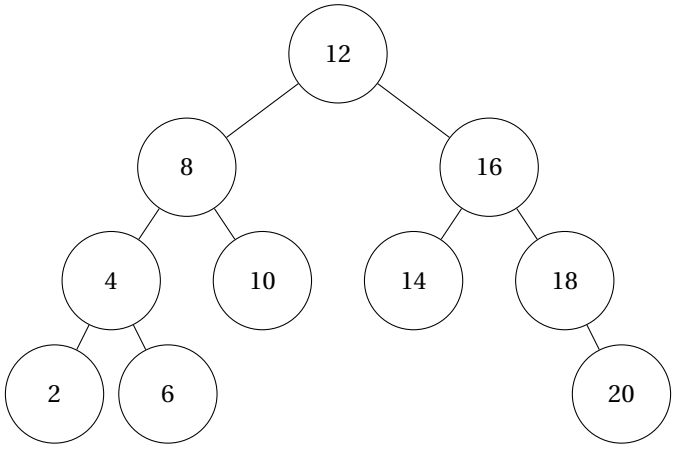

Figura 2: `numHFtree1`

11. *Altura de BST*
    Calcula la altura de un árbol binario de búsqueda, definida como la longitud del camino más largo desde la raíz hasta alguna hoja. Se recomienda utilizar recursión para comparar las alturas de los subárboles.

```haskell
bTreeHeight :: BTree a -> Int
```

Ejemplo usando *numHTree1*:

```
ghci> bTreeHeight numHTree1
4
```

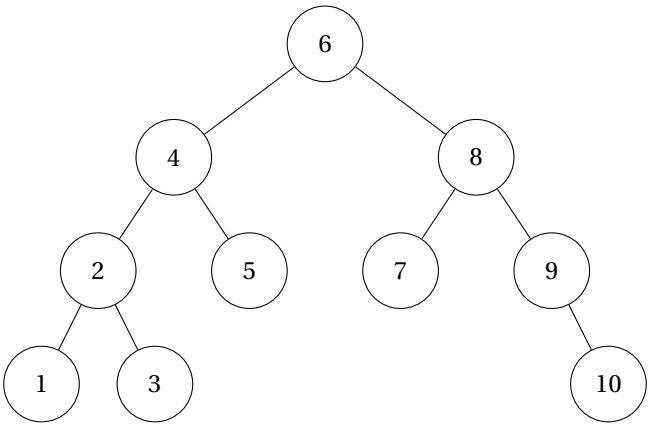

Figura 3: `numHtree1`

Entrega

1. La tarea se entrega en *Classroom*.
2. La tarea se entrega en equipos de hasta **3** 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 de la biblioteca `Data.List` (`https://hackage.haskell.org/package/base-4.21.0.0/docs/Data-List.html`), 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.
12. **Réplica**: La práctica está dividida en dos partes
    - *Tarea para casa* que representará el 80 % de la calificación.
    - *Resolución en clase* deberá resolverse en el laboratorio antes de finalizar la clase, con un valor del 20 %.

    Ambas actividades estarán sujetas a una defensa, por lo que todas las personas que conformen el equipo deberán demostrar comprensión del código y justificar sus decisiones técnicas durante la evaluación.