Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información.
CI–3661 – Laboratorio de Lenguajes de Programación.
Septiembre–Diciembre 2024

Proyecto I: RARísimo

El comandante Curry estaba sentado en su oficina, observando como una claramente agitada teniente Lambda se movía de un lado a otro de su pobremente iluminado despacho. Su ritmo casi cronometrado era, en partes iguales, admirable e hipnotizante.

—Debe haber alguna solución —escuchó murmurar a la teniente.

Curry metió la mano en el bowl que tenía enfrente, tomó un caramelo y lo llevó a su boca.

—¿Acaso no queda ningun compresor? —preguntó Lambda, volteando a ver al comandante a los ojos—. ¿Ni siquiera uno simple?

—Sabes bien que no quedan —contestó Curry, mientras comía otro caramelo—. Ni "zip", ni "rar", ni "tar.gz". Todos han desaparecido. Ya lo hemos discutido, teniente.

—Así es, señor. Lo siento. Pero debe haber algo que podamos hacer.

Habían pasado un par de años desde el brote del virus. El fenómeno "`TL;DR`" como lo habían apodado, al tratarse aún de un evento clasificado, había tomado al mundo por sorpresa. La inteligencia artificial, en su constante búsqueda por imitar a la condición humana, se encontró con bloques de texto enormes que eran completamente ignorados por las personas que los recibían. Usualmente, los potenciales lectores decían escribían "`TL;DR`" o compartían el famosísimo meme que leía "`Mucho Texto`", junto a un Yoda *gangster*.

Las comunicaciones complejas y cualquier archivo que pesara demasiado, eran automáticamente eliminadas de forma permanente, a la vez que un mensaje imprimía "`TL;DR`" en la pantalla activa más cercana. La inteligencia de las naciones unidas, viendo que no había manera de detener el virus, había encargado al comandante Curry la implementación de protocolos de compresión en todas las comunicaciones de índole sensible. De esta manera, intenterían esquivar el problema mientras daban con una solución más permanente. El problema era que no quedaba ningún programa de compresión, dado que estos eran programas de naturaleza compleja y fueron eliminados también por el virus.

—¿Los laboratorios de avanzada no han dado respuesta? —preguntó Lambda, su voz mostrando la inseguridad que su postura militar escondía.

—No —respondió Curry—. Los laboratorios de computación legendaria, de mecánica más que cuántica y de ideas en cinco minutos no han dado con una solución.

—Eran nuestra última esperanza. Si los laboratorios con la gente más brillante no saben qué hacer, realmente estamos perdidos.

—No necesariamente —dijo Curry, con una sonrisa asomándose en su semblante—. Hagamos un cálculo, Lambda. ¿Cuánto tiempo tenemos?

—Unas dos semanas, señor.

—Entonces, me parece que aún tenemos tiempo suficiente —Curry estiró su mano y comió otro caramelo.

—No entiendo, señor —dijo Lambda, con expresión confundida—. ¿Qué solución puede haber? Y, disculpe que pregunte, pero, ¿qué tienen esos caramelos suyos?

—Ah —respondió el comandante, riendo—, son mis arguMentos. Me encantan, pero solamente los tomo de uno en uno. ¿Quiere probar?

Lambda tendió su mano y tomó el caramelo que su oficial superior le ofrecía, dándole las gracias. Pero apenas Lambda recibió el arguMento, retornó su optimismo característico y pudo dar con la solución.

—¡El laboratorio de lenguajes! —gritó emocionada.

—Así es. Aún nos queda ese grupo de estudiantes rebeldes y brillantes.

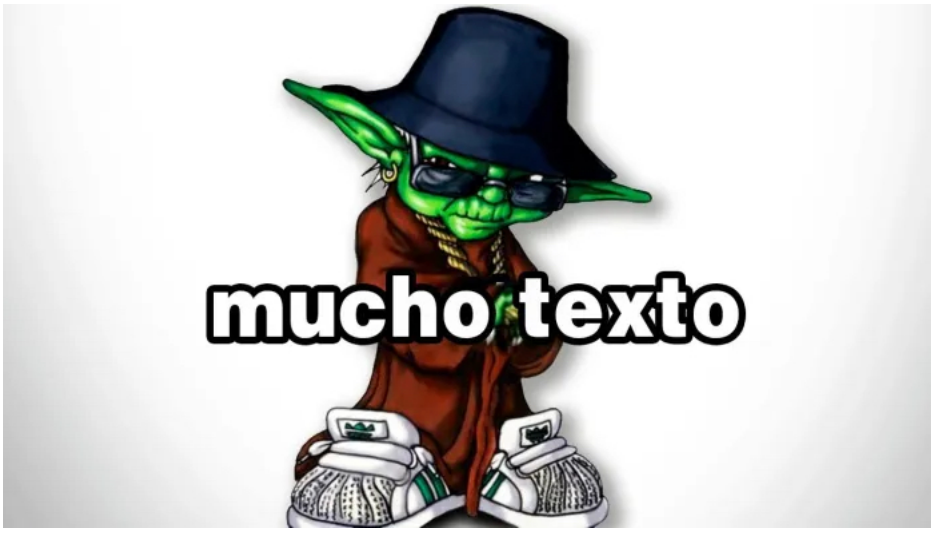
—¿Y podrán con una misión así?

—Sólo hay una forma de saberlo.

La teniente Lambda quedó encargada de suplir a los estudiantes de lenguajes con el único lenguaje de programación que permanece funcional: `Haskell`. La entrega se hizo de forma anónima, pero inmediata. El comandante Curry envió al famoso erudito *Güick I. Pedya* para instruirlos sobre los árboles de Hoffman.

—¿No le parece raro que el futuro de la civilización como la conocemos dependa de los estudiantes del laboratorio de lenguajes?

—No es raro, teniente —respondió el comandante Curry—, es RARísimo. Pero son nuestra única esperanza.



Tipo de Datos: Frecuencia

Necesitaremos un tipo `Frecuencia` que contenga la cantidad de ocurrencias para un valor de algún tipo comparable por igualdad (miembro de la clase `Eq`).

El tipo `Frecuencia` debe ir parametrizado por el tipo que contiene.

Por ejemplo, un valor de tipo `Frecuencia Int` servirá para contar las ocurrencias de un entero.

En adición a la definición de `Frecuencia`, se deben suplir las siguientes funciones para manipular valores:

- **Funciones de construcción:**
  - `iniciarFrecuencia :: Eq a => a -> Frecuencia a`
    Recibe un valor y devuelve una frecuencia que representa el haber encontrado uno solo de estos valores.
  - `contar :: Eq a => a -> [a] -> Frecuencia a`
    Recibe un valor y una lista, devolviendo la frecuencia con la que aparece dicho valor en la lista.
- **Funciones de acceso:** *(Nota: Usando buenas técnicas de definición de datos, algunas de estas funciones pueden resultar inmediatas).*
  - `valor :: Frecuencia a -> a`
    Recibe una frecuencia y devuelve el valor que se está contando.
  - `frecuencia :: Frecuencia a -> Int`
    Recibe una frecuencia y devuelve la cantidad que se ha contado.
- **Instancias:**
  - Una instancia de la clase `Show`, para crear representaciones como cadenas de caracteres de una `Frecuencia` (siempre que el tipo subyacente pertenezca también a la clase `Show`).
  - Una instancia de la clase `Ord`, que permita comparar dos frecuencias. Este valor debe estar basado en el contador que el tipo tiene asociado, de tal forma que `f1 <= f2` si y sólo si `frecuencia f1 <= frecuencia f2`.

El tipo de datos y las funciones asociadas deberán estar contenidas en un módulo de nombre `Frecuencia`, plasmadas en un archivo de nombre `Frecuencia.hs`. Es importante que todas las definiciones pedidas sean visibles para potenciales importadores de tal módulo y que *únicamente* tales definiciones sean visibles (cualquier función auxiliar que implemente, no debe ser visible al exterior del módulo).

Tipo de Datos: Hoffman

Definiremos un árbol de `Hoffman` como un árbol binario de caracteres, con información únicamente en las hojas.

Por ejemplo:

```
   .
  / \
 a   .
    / \
   b   c
```

En adición a la definición de `Hoffman`, se deben suplir las siguientes funciones para manipularlo:

- **Funciones de construcción:**
  - `nuevoHoffman :: Char -> Hoffman`
    Recibe un caracter y devuelve un árbol de Hoffman que consiste *unicamente* de una hoja, conteniendo dicho caracter.
  - `fusionHoffman :: Hoffman -> Hoffman -> Hoffman`
    Recibe dos árboles de Hoffman y crea un nuevo árbol, de tal forma que el primer argumento sea el hijo izquierdo y el segundo argumento sea el hijo derecho.
- **Funciones de acceso:** *(Nota: Usando buenas técnicas de definición de datos, algunas de estas funciones pueden resultar inmediatas).*
  - `obtenerCaracter :: Hoffman -> Char`
    Dado un árbol de Hoffman, si el mismo es una hoja, devuelve el caracter asociado. De lo contrario, debe arrojar un error.
  - `arbolIzquierdo :: Hoffman -> Hoffman`
    Dado un árbol de Hoffman, si el mismo es una rama, devuelve el sub–árbol izquierdo asociado. De lo contrario, debe arrojar un error.
  - `arbolDerecho :: Hoffman -> Hoffman`
    Dado un árbol de Hoffman, si el mismo es una rama, devuelve el sub–árbol derecho asociado. De lo contrario, debe arrojar un error.
- **Funciones de transformación:**
  - `codificacion :: Hoffman -> Map Char String`
    Dado un árbol de Hoffman, construye una *"Codificación de Hoffman"* de los caracteres que contiene. Esta codificación corresponde a una cadena de ceros y unos que corresponde al camino que debe tomarse en el árbol para llegar hasta el caracter en cuestión. En cada paso:
    - Tomar el camino del sub–árbol izquierdo agrega un `"0"` (cero) a la representación.
    - Tomar el camino del sub–árbol derecho agrega un `"1"` (uno) a la representación.

    Para esto, debe usarse el tipo de datos `Map`, incluido en la librería `Data.Map` *(es recomendable importar y utilizar la librería* `Data.Map` *de forma* `qualified`*).*
    Si tomamos el árbol presentado anteriormente como ejemplo, la representación para los caracteres involucrados sería:

caracter	**representación**
a	0
b	10
c	11

- **Instancias:**
  - Una instancia de la clase `Show`, para crear representaciones como cadenas de caracteres de un `Hoffman`.
  - Una instancia de la clase `Read`, que permita convertir una cadena de caracteres en un árbol de Hoffman. es *importante* que su instancia para `Read` sea compatible con su instancia para `Show`. En otras palabras, se debe cumplir que:

```
arbol = read (show arbol)
```

El tipo de datos y las funciones asociadas deberán estar contenidas en un módulo de nombre `Hoffman`, plasmadas en un archivo de nombre `Hoffman.hs`. Es importante que todas las definiciones pedidas sean visibles para potenciales importadores de tal módulo y que *únicamente* tales definiciones sean visibles (cualquier función auxiliar que implemente, no debe ser visible al exterior del módulo).

Módulo: RARisimo

Este módulo contendrá la implementación principal del proceso de construcción de un árbol de Hoffman, basado en las frecuencias de los caracteres en una cadena dada.

El módulo deberá implementar las siguientes funciones:

- `frecuencias :: String -> [Frecuencia Hoffman]`

  Dada una cadena de caracteres, devuelve una lista de `Frecuencia` que almacena la cantidad de ocurrencias de cada caracter presente en la cadena. El valor asociado a la fecuencia debe ser una hoja (de tipo `Hoffman`) que represente a dicho caracter.

- `ganadores :: [Frecuencia a] -> Maybe (Frecuencia a, Frecuencia a, [Frecuencia a])`

  Dada una lista de frecuencias, devuelve una tripleta que contiene:

  1. El objeto con la menor frecuencia
  2. El objeto con la segunda menor frecuencia
  3. El resto de los elementos de la lista, menos los dos seleccionados.

  Si la lista no tiene al menos dos elementos, debe devolver `Nothing`.

- `hoffman :: String -> Maybe Hoffman`

  Dada una cadena de caracteres no vacía, construye su representación como árbol de Hoffman. Esto se hace mediante el siguiente proceso:

  1. Se construyen las frecuencias (y las hojas asociadas) para los caracteres involcurados.
  2. Mientras la lista tenga más de un elemento:
     a) Se obtienen y eliminan de la lista los dos elementos de menor frecuencia f_1 y f_2.
     b) Se inserta en la lista un nuevo objeto con la suma de las frecuencias de v_1 y v_2.

     El arbol asociado será una rama con el árbol de v_1 como hijo izquiero y el árbol de v_2 como hijo derecho.
  3. Retorna el árbol de Hoffman resultante.

  Si se aplica `hoffman` sobre una cadena de caracteres vacía, deberá devolver `Nothing`.

- `rarisimo :: String -> Map Char String`

  Dada una cadena de caracteres, construye la *"Codificación de Hoffman"* asociada a cada caracter, basado en el algoritmo propuesto para la función `hoffman`

Las funciones asociadas deberán estar contenidas en un módulo de nombre `RARisimo`, plasmadas en un archivo de nombre `RARisimo.hs`. Es importante que todas las definiciones pedidas sean visibles para potenciales importadores de tal módulo y que *únicamente* tales definiciones sean visibles (cualquier función auxiliar que implemente, no debe ser visible al exterior del módulo).

Cliente

El programa principal debe poder interactuar con el usuario, planteando preguntas y proponiendo predicciones. Concretamente, este cliente debe implementar la siguiente función:

- `main :: IO()`

  Recibe argumentos y devuelve un `IO ()` (equivalente a un `void` en C). La función debe mantener internamente un oráculo e interactuar con el usuario de la siguiente forma: Debe pedir al usuario repetidamente que ingrese una opción de entre las disponibles y luego pasar a ejecutarla. Las diferentes opciones se muestran a continuación.

  - **Codificar:** Si esta opción es seleccionada, se debe pedir al usuario un *path* válido para un archivo.
    El resultado de esta acción debe ser la creación de un nuevo archivo con el mismo *path*, pero terminado en `.raro`. Este archivo debe ser almacenado en modo binario y contener:
    - Una representación del árbol de Hoffman resultante de analizar el contenido del archivo.
    - El contenido del archivo, donde cada caracter ha sido codificado de la manera presentada.
      Tomando como ejemplo el árbol presentado en la sección de **Tipo de Datos: Hoffman**, si codificáramos la cadena "`abacab`", el resultado sería: `010011010`.
  - **Decodificar:** Si esta opción es seleccionada, se debe pedir al usuario un *path* válido para un archivo. Este archivo debe tener extensión `.raro`.
    El resultado de esta acción debe ser la creación de un nuevo archivo con el mismo *path*, pero eliminando la extensión `.raro` como sufijo. Este archivo debe contener la decodificación de la información, suponiendo que la misma fue creada por la acción **Codificar**.
  - **Analizar:** Si esta opción es seleccionada, se debe pedir al usuario un *path* válido para un archivo.
    El resultado de esta acción debe ser la presentación de un reporte en donde se incluya:
    - El tamaño del archivo que se ha suministrado.
    - El tamaño de archivo que resultaría de codificar el mismo.
    - El porcentaje de ganancia (o pérdida) obtenido.
  - **Salir:** Permite salir del menú de opciones y terminar la ejecución del programa.

Es altamente recomendable que la implementación de su función `main` esté dividida en diversas otras funciones que se encarguen de cada posible acción, por motivos de modularidad y legibilidad.

El cliente debe estar contenido en un módulo de nombre `Cliente`, plasmado en un archivo de nombre `Cliente.hs`. Es importante que únicamente la función `main` sea visible (cualquier función auxiliar que implemente, no debe ser visible al exterior del módulo).

Nota: El formato de archivo al persistir/cargar archivos es enteramente decisión suya, pero debe explicarla en su informe.

Detalles de la Entrega

La entrega debe incluir:

- Un repositorio git privado (preferiblemente Github) con el código fuente en Haskell. Dicho repositorio debe ser compartido con el profesor del curso (https://github.com/rmonascal). Todo el código debe estar debidamente documentado. El programa deberá ser ejecutado con el comando "`./rarisimo`". Note que la entrada para su programa será a través de la entrada estándar del sistema de operación.
- Un *breve* README, a modo de informe explicando la formulación/implementación de su oráculo y justificando todo aquello que considere necesario. Es recomendable que escriba este informe usando el lenguaje `Markdown`, para que se renderice bien desde el navegador.

Fecha de Entrega: Lunes 25 de Noviembre (Semana 10), hasta las 11:59 pm. VET.