Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información.
CI–3661 – Laboratorio de Lenguajes de Programación.
Enero–Marzo 2023

Tarea 1: Programación Funcional y Haskell

(15 pts)

Implementación:

1. (3 pts) – Considere la estructura de datos `Conjunto`, que representa conjuntos potencialmente infinitos. Para ser capaces de conocer la pertenencia de elementos en dichos conjuntos (aún siendo infinitos) los mismos no deben representarse como enumeraciones explícitas de sus elementos, si no como la función que sabe distinguir los elementos que pertenecen al mismo (función característica del conjunto). Por ejemplo, el conjunto de los números enteros pares puede representarse como la función: `(\num -> even num)`.

   A continuación se presenta la definición del tipo de datos `Conjunto` en Haskell:

   ```
   type Conjunto a = a -> Bool
   ```

   Tomando en cuenta la definición anterior, debe implementar entonces cada una de las siguientes funciones (válidas independientemente del tipo concreto que tome `a` y sin cambiar sus firmas):

   a) `miembro :: Conjunto a -> a -> Bool`
   Debe devolver la pertenencia en el conjunto proporcionado, de un elemento dado.

   b) `vacio :: Conjunto a`
   Debe devolver un conjunto vacío.

   c) `singleton :: (Eq a) => a -> Conjunto a`
   Debe devolver un conjunto que contenga únicamente al elemento proporcionado.

   d) `desdeLista :: (Eq a) => [a] -> Conjunto a`
   Debe devolver un conjunto que contenga a todos los elementos de la lista proporcionada.

   e) `complemento :: Conjunto a -> Conjunto a`
   Debe devolver un conjunto que contenga únicamente todos los elementos que no estén en el conjunto proporcionado (pero que sean del mismo tipo).

   f) `union :: Conjunto a -> Conjunto a -> Conjunto a`
   Debe devolver un conjunto que contenga todos los elementos de cada conjunto proporcionado.

g) `interseccion :: Conjunto a -> Conjunto a -> Conjunto a`

Debe devolver un conjunto que contenga solo los elementos que estén en los dos conjuntos proporcionados.

h) `diferencia :: Conjunto a -> Conjunto a -> Conjunto a`

Debe devolver un conjunto que contenga los elementos del primer conjunto proporcionado, que no estén en el segundo.

i) `transformar :: (b -> a) -> Conjunto a -> Conjunto b`

Dada una función f y un conjunto A, devuelva un conjunto B tal que A es el resultado de aplicar f a todos los elementos de B. Por ejemplo, si la función f es sumar 1 y el conjunto A es $\{2, 4, 8\}$, un conjunto resultado podría ser $\{1, 3, 7\}$ (nótese que tal conjunto podría no ser único, por lo que se espera que retorne cualquiera de ellos, en caso de existir varias opciones).

2. (3 pts) – Considere el tipo de datos `ArbolMB`, que se define a continuación:

```
data ArbolMB a = Vacio
               | RamaM a (ArbolMB a)
               | RamaB a (ArbolMB a) (ArbolMB a)
```

a) Los constructores de un tipo de datos pueden verse como funciones que reciben (currificados) los argumentos original del mismo y arrojan como resultado un valor del tipo en cuestión.

Por ejemplo:
```
RamaM :: a -> ArbolMB a -> ArbolMB a
```

Diga los tipos correspondientes a los constructores `Vacio` y `RamaB`, vistos como funciones.

b) Se desea implementar una función que transforme valores de tipo `(ArbolMB a)` en algún otro tipo `b`. Claramente, dicha función debe tener tres casos (uno por cada constructor). Implementaremos cada caso como una función aparte: `transformarVacio`, `transformarRamaM` y `transformarRamaB`, respectivamente. Cada una de estas funciones debe tomar los mismos argumentos que los constructores respectivos. Sin embargo, la transformación se hará a profundidad, por lo que se puede suponer que cada argumento de tipo `(ArbolMB a)` ya ha sido transformado a `b`.

Por ejemplo:
```
transformarRamaM :: a -> b -> b
```

Diga los tipos correspondientes a las funciones transformadoras `transformarVacio` y `transformarRamaB`.

c) Implementaremos ahora la función transformadora deseada, tomando como argumentos las tres funciones que creamos en la parte (b). Llamaremos a esta función `plegarArbolMB` y su firma *(la cual debe completar con su respuesta a la parte (b))* sería la siguiente:

```
plegarArbolMB :: ( ¿? )          -- El tipo de transformarVacio
              -> (a -> b -> b)   -- El tipo de transformarRamaM.
              -> ( ¿? )          -- El tipo de transformarRamaB.
              -> ArbolMB a       -- El arbol a plegar.
              -> b               -- El resultado del plegado.
```

Complete la definición de la función `plegarArbolMB`, propuesta a continuación, recordando que las transformaciones deben hacerse a profundidad para poder garantizar un valor transformado como argumento a las diferentes funciones *(nótese que la función auxiliar* `plegar` *recibe implícitamente el valor de tipo* `(ArbolMB a)` *a considerar).*

```
plegarArbolMB transVacio transRamaM transRamaB = plegar
    where
        plegar Vacio         = ¿?
        plegar (RamaM x y)   = transRamaM x (plegar y)
        plegar (RamaB x y z) = ¿?
```

d) Usando nuestra función `plegarArbolMB`, se desea implementar ahora una función `sumaArbolMB`, que dado un valor de tipo `(Num a) => ArbolMB a` calcule y devuelva la suma de todos datos almacenados en el tipo.
Complete la definición de la función `sumarArbolMB`, propuesta a continuación, usando únicamente una llamada a `plegarArbolMB` y definiendo las funciones de transformación necesarias. *(nótese que la función* `plegarArbolMB` *recibe implícitamente el valor de tipo* `((Num a) =>ArbolMB a)` *a considerar).*

```
sumarArbolMB :: (Num a) => ArbolMB a -> a

sumarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
    where
        transVacio = ¿?
        transRamaM = ¿?
        transRamaB = ¿?
```

e) Usando nuevamente nuestra función `plegarArbolMB`, se desea implementar ahora una función `aplanarArbolMB`, que dado un valor de tipo `ArbolMB a` calcule y devuelva una sola lista con todos los elementos contenidos en la estructura. En el caso de la rama con un solo hijo, el elemento debe ir antes que los elementos del hijo. En el caso de la rama con dos hijos, los elementos del primer hijo deben ir antes que el elemento de la rama y este, a su vez, debe ir antes que los elementos del segundo hijo (un recorrido en *in–order*).
Complete la definición de la función `aplanarArbolMB`, propuesta a continuación, usando únicamente una llamada a `plegarArbolMB` y definiendo las funciones de transformación necesarias. *(nótese que la función* `plegarArbolMB` *recibe implícitamente el valor de tipo* `(ArbolMB a)` *a considerar).*

```
aplanarArbolMB :: ArbolMB a -> [a]

aplanarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
    where
        transVacio = ¿?
        transRamaM = ¿?
        transRamaB = ¿?
```

f) Usando nuevamente nuestra función `plegarArbolMB`, se desea implementar ahora una función `analizarArbolMB`, que dado un valor de tipo `(Ord a) => ArbolMB a` calcule y devuelva *posiblemente* una tupla con 3 elementos. El primero debe ser el mínimo elemento presente en la estructura, el segundo debe ser el máximo y el tercero debe ser un booleano que sea cierto si y solo si la lista que resultaría de llamar a la función `aplanarArbolMB` estaría ordenada de menor a mayor (esto es, que sea un árbol de búsqueda). En el caso de un valor de tipo `Vacio`, se debe devolver el valor `Nothing`. *(Pista: No es conveniente llamar explícitamente a la función* `aplanarArbolMB` *para calcular el 3er elemento de la tupla.)*
Complete la definición de la función `analizarArbolMB`, propuesta a continuación, usando únicamente una llamada a `plegarArbolMB` y definiendo las funciones de transformación necesarias. *(nótese que la función* `plegarArbolMB` *recibe implícitamente el valor de tipo* `((Ord a) =>ArbolMB a)` *a considerar).*

```
analizarArbolMB :: (Ord a) => ArbolMB a -> Maybe (a, a, Bool)

analizarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
    where
        transVacio = ¿?
        transRamaM = ¿?
        transRamaB = ¿?
```

g) Considere ahora un tipo de datos más general `Gen a`, con n constructores diferentes. ¿Si se quisiera crear una función `plegarGen`, con un comportamiento similar al de `plegarArbolMB`, cuantas funciones debe tomar como argumento (además del valor de tipo `Gen a` que se desea plegar)?

h) Considere ahora el caso especial donde hay 2 posibles constructores.

```
data [a] = (:) a [a]
         | []
```

¿Qué función predefinida sobre listas, en el Preludio de Haskell, tiene una firma y un comportamiento equivalente al de implementar una función de plegado para el tipo propuesto?

3. (3 pts) – Los monads son estructuras que representan cómputos con algún comportamiento particular, encapsulando la implementación del mismo en las definiciones de sus funciones `>>=` y `return`. Por ejemplo: el monad `Maybe` representa cómputos que pueden fallar, el monad `[]` representa cómputos no–deterministas y el monad `IO` representa cómputos impuros.

   Se desea implementar entonces un monad que represente cómputos secuenciales. Es decir, dado un estado inicial (por ejemplo, valor de variables en el alcance) se debe obtener un resultado final para el cómputo y un nuevo estado (resultado de posibles alteraciones al estado inicial). Notemos entonces que un cómputo secuencial en realidad puede verse como un alias para una función `s -> (a, s)`, donde `s` es el tipo del estado y `a` el tipo del resultado.

   Construyamos un tipo de datos entonces para representar cómputos secuenciales.

```
newtype Secuencial s a = Secuencial (s -> (a, s))
```

   De la definición anterior debemos notar dos cosas: el identificador `Secuencial` es usado tanto como nombre de tipo como constructor; la notación `newtype` se ha utilizado pues solo existe un posible constructor con un solo argumento. Por lo tanto, dicho argumento es equivalente en contenido al tipo completo, pero conviene no hacerlo un alias para que los tipos no se mezclen (no pasar funciones cualesquiera como cómputos secuenciales).

   Queremos que nuestro tipo sea un monad, por lo que haremos una instancia para él.

```
instance Monad (Secuencial s) where ...
```

   a) ¿Por qué se tomó `(Secuencial s)` como la instancia para el monad y no simplemente `Secuencial`?

   b) Diga las firmas para las funciones `return`, `>>=`, `>>` y `fail` para el caso especial del monad `(Secuencial s)`

   c) Implemente la función `return` de tal forma que *inyecte* el argumento pasado como argumento, dejando el estado inicial intacto. Esto es, dado un estado inicial, el resultado debe ser el argumento pasado junto al estado inicial sin cambios.

   d) Complete la implementación de la función `>>=` que se da a continuación:

```
(Secuencial programa) >>= transformador =
    Secuencial $ \estadoInicial ->
        let (resultado, nuevoEstado)   = programa      ¿?
            (Secuencial nuevoPrograma) = transformador ¿?
        in nuevoPrograma ¿?
```

   (Pista: Ayúdese con los tipos esperados y la intuición para dar un valor a cada una de las interrogantes, las cuales corresponderan–cada una–a un solo identificador de los previamente definidos.)

   e) Demuestre las tres leyes monádicas aplicadas al tipo `Secuencial s`.
   Recuerde que las tres leyes monádicas son:

   - **Identidad izquierda**: `return a >>= h == h a`
   - **Identidad derecha**: `m >>= return == m`
   - **Asociatividad**: `(m >>= g) >>= h == m >>= (\x -> g x >>= h)`

Investigación:

1. (3 pts) – La programación funcional está basada en el Lambda Cálculo, propuesto por Alonzo Church. Dicho cálculo está basado en llamadas λ–*expresiones*. Estas expresiones toman una de tres posibles formas:

x	–	donde x es un identificador.
$\lambda x\ .\ E$	–	donde x es un identificador y E es una λ–*expresión*, es llamada λ–*abstracción*.
$E\ F$	–	donde E y F son λ–*expresiones*, es llamada *aplicación funcional*.

   Se dice que una λ–*expresión* está normalizada si para cada sub–expresión que contiene, correspondiente a una aplicación funcional, la sub–expresión del lado izquierdo no evalúa a una λ–*abstracción*. De lo contrario, dicha sub–expresión aún podría evaluarse. A continuación se presenta una semántica formal simplificada para la evaluación de λ–*expresiones*, suponiendo que no existe el problema de captura de variable (ninguna variable libre resulta ligada a una abstracción):

   $$\begin{array}{lcl} eval(x) & = & x. \\ eval(\lambda x\ .\ E) & = & \lambda x.eval(E) \\ eval(x\ F) & = & x\ eval(F) \\ eval((\lambda x\ .\ E)\ F) & = & eval(E)\ [x := eval(F)] \\ eval((E\ F)\ G) & = & eval(eval(E\ F)\ eval(G)) \end{array}$$

   Tomando esta definición en cuenta, conteste las siguientes preguntas:

   a) ¿Cual es la forma normalizada para la expresión: $(\lambda x\ .\ \lambda y\ .\ x\ y\ y)\ (\lambda z\ .\ z\ O)\ L$?

   b) Considere una aplicación funcional, de la forma $E\ F$. ¿Existen posibles expresiones E y F, tal que el orden en el que se evalúen las mismas sea relevante (arroje resultados diferentes)? De ser así, proponga tales expresiones E y F. En cualquier caso, justifique su respuesta.

   c) Considere una evaluación para una λ–*expresión* de la forma $((\lambda x\ .\ E)\ F)$. ¿Qué cambios haría a la semántica formal de la función *eval* para este caso, si se permitiesen identificadores repetidos? [Considere, como ejemplo, lo que ocurre al evaluar la siguiente expresión: $(\lambda x\ .\ (\lambda y\ .\ x\ y))\ y$]

2. (3 pts) – Considere la siguientes funciones:

```
id :: a -> a
id x = x

const :: a -> b -> a
const x _ = x

subs :: (a -> b -> c) -> (a -> b) -> a -> c
subs x y z = x z (y z)
```

La primeras dos funciones son parte del preludio de Haskell.

a) Evalúe la expresión: `subs (id const) const id`. No evalúe la expresión en Haskell, pues si el resultado es una función no podrá imprimirlo. Evalúe la expresión a mano y exponga el resultado en término de las funciones antes propuestas (utilice evaluación normal: primero la función, luego los argumentos).

b) Proponga una expresión (únicamente compuesta por las funciones definidas anteriormente), que no esté en forma normal, cuya evaluación resulte en la misma expresión y por lo tanto nunca termine.

c) Reimplemente la función `id` en términos de `const` y `sub`. *(Pista: puede utilizar el tipo unitario* () *para representar un argumento del cual no importa su valor, pero que igual debe ser pasado como parámetro a una función.)*

d) Discuta la relación entre las funciones propuestas y el cálculo de combinadores SKI.

Detalles de la Entrega

La entrega de la tarea consistirá de un único archivo PDF con su implementación para las funciones pedidas y respuestas para las preguntas planteadas.

La tarea deberá ser entregada al prof. Ricardo Monascal *únicamente* a su dirección de correo electrónico: (rmonascal@gmail.com) a más tardar el Domingo 19 de Noviembre, a las 11:59pm. VET.