Universidad Simón Bolívar

Departamento de Computación y Tecnología de la Información.

CI–3661 – Laboratorio de Lenguajes de Programación.

Enero–Marzo 2020

Integrantes: José Ramón Barrera Melchor **carnet:** 15-10123

Carlos Rafael Rivero Panades **carnet:** 13-11216

Tarea II: Haskell (10 pts).

Implementación:

0. (3.5 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) (0.25 pts) – 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.

```
Vacio  :: ArbolMB a
RamaM :: a -> ArbolMB a -> ArbolMB a
RamaB  :: a -> ArbolMB a -> ArbolMB a -> ArbolMB a
```

b) (0.25 pts) – 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.

```
transformarVacio  :: a -> b
transformarRamaM :: a -> b -> b
transformarRamaB  :: a -> b -> b -> b
```

c) (1 pt) – 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) = ¿?
```

```
plegarArbolMB  :: (a -> b)
               -> (a -> b -> b)
               -> (a -> b -> b -> b)
               -> ArbolMB a
               -> b
plegarArbolMB transVacio transRamaM transRamaB = plegar
where
    plegar Vacio          = transVacio
    plegar (RamaM x y)    = transRamaM x (plegar y)
    plegar (RamaB x y z)  = transRamaB x (plegar y) (plegar z)
```

d) (0.5 pts) – 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    = ¿?
```

```
sumarArbolMB :: (Num a) => ArbolMB a -> a
sumarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
where
    transVacio          = 0
    transRamaM x y    = x + sumarArbolMB y
    transRamaB x y z  = x + sumarArbolMB y + sumarArbolMB z
```

e) (0.5 pts) – 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  = ¿?
```

```
aplanarArbolMB :: ArbolMB a -> [a]
aplanarArbolMB   = plegarArbolMB transVacio transRamaM transRamaB
where
        transVacio           = []
        transRamaM x y    = x : aplanarArbolMB y
        transRamaB x y z  = aplanarArbolMB y : x : aplanarArbolMB z
```

f) (0.5 pt) – 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 = ¿?
```

```
import Data.Tuple.Select
import Data.Maybe
analizarArbolMB :: (Ord a) => ArbolMB a -> Maybe (a, a, Bool)
analizarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
where
        transVacio              = Nothing
        transRamaM x y    = Just  a, b, c where
                a = min x (Sel1 d) where
                        d = (fromMaybe ((maxBound :: Int), 0, True) (analizarArbolMB y))
                b = max x (Sel2 d) where
                        d = (fromMaybe (0, (minBound :: Int), True) (analizarArbolMB y))
                c = (Sel3 d) && (x >= (Sel2 d)) where
                        d = (fromMaybe (0, (minBound :: Int), True) (analizarArbolMB y))
        transRamaB x y z  = Just  a, b, c where
                a = min x (min (Sel1 d) (Sel1 e)) where
                        d = (fromMaybe ((maxBound :: Int), 0, True) (analizarArbolMB y))
                        e = (fromMaybe ((maxBound :: Int), 0, True) (analizarArbolMB z))
                b = max x (max (Sel2 d) (Sel2 e)) where
                        d = (fromMaybe (0, (minBound :: Int), True) (analizarArbolMB y))
                        e = (fromMaybe (0, (minBound :: Int), True) (analizarArbolMB z))
                c = (Sel3 d) && (x >= (Sel2 d)) && (x <= (Sel1 e)) where
                        d = (fromMaybe (0, (minBound :: Int), True) (analizarArbolMB y))
                        e = (fromMaybe ((maxBound :: Int), 0, True) (analizarArbolMB z))
```

g) (0.25 pts) – 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)?

Debería tomar n funciones.

h) (0.25 pts) – Considere ahora el caso especial donde hay 2 posibles constructores.

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

¿Que 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?

foldr

i) (Extra) – ¿Qué typeclass de Haskell incluye a las estructuras que permiten este comportamiento?

foldable

1. (3.5 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) (0.5 pts) – ¿Por qué se tomó (Secuencial s) como la instancia para el monad y no simplemente Secuencial?

Es necesario poner la S en la instancia porque el Monad Secuencial depende de su estado interno para generar el resultado final. Y porque la instancia debe ser de kind (* -> *)

b) (1 pt) – Diga las firmas para las funciones return, >>=, >> y fail para el caso especial del monad (Secuencial s)

```
return :: a -> Secuencial s a
(>>=) :: Secuencial s a -> (a -> Secuencial s b) -> Secuencial s b
(>>) :: Secuencial s a -> Secuencial s b -> Secuencial s b
fail :: String -> Secuencial s a
```

c) (1 pt) – 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.

```
return x = Secuencial ( \s -> (x, s) )
```

d) (1 pt) – 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.)

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

e) (Extra) – ¿Conoce algún monad de Haskell que tenga un comportamiento similar? ¿Cuál?

```
State
```

(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) (0.5 pts) – 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).

```
        subs (id const) const id
===
        (id const) id (const id)
===
        const id (const id)
===
        id
```

b) (0.5 pts) – 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.

```
subs subs const ( subs ( const ( subs subs ( subs ( subs subs const ) ) ) ) const )
```

c) (1 pt) – 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.)

```
id :: a -> a
id x = const x ()
```

d) (1 pt) – Discuta la relación entre las funciones propuestas y el cálculo de combinadores SKI.

El cálculo combinatorio SKI es una lógica combinatoria, un sistema computacional que puede ser percibido como una versión reducida del cálculo lambda sin tipo. En este se encuentran definidas las siguientes operaciones:

Ix = x

Kxy = x

Sxyz = xz(yz)

Con lo cual podemos establecer claras equivalencias entre estos y las funciones propuestas, a saber:

```
S = subs
K = const
I = id
```

Ski