Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información.
CI–3661 – Laboratorio de Lenguajes de Programación.
Septiembre–Diciembre 2012

Tarea II: Haskell – *Solución*

(10 pts)

Implementación:

0. (3.5 pts) – Considere el tipo de datos `Origami`, que se define a continuación:

```
data Origami a = Papel
               | Pico a (Origami a)
               | Valle a (Origami a)
               | Compuesto (Origami a) (Origami 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:
```
Pico :: a -> Origami a -> Origami a
```

Diga los tipos correspondientes a los constructores `Papel`, `Valle` y `Compuesto`, vistos como funciones.

```
Papel :: Origami a
```

```
Valle :: a -> Origami a -> Origami a
```

```
Compuesto :: Origami a -> Origami a -> Origami a
```

b) (0.25 pts) – Se desea implementar una función que transforme valores de tipo `(Origami a)` en algún otro tipo `b`. Claramente, dicha función debe tener cuatro casos (uno por cada constructor). Implementaremos cada caso como una función aparte: `transformarPapel`, `transformarPico`, `transformarValle` y `transformarCompuesto` 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 `(Origami a)` ya ha sido transformado a `b`.

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

Diga los tipos correspondientes a las funciones transformadoras `transformarPapel`, `transformarValle` y `transformarCompuesto`.

```
transformarPapel :: b
```

```
transformarValle :: a -> b -> b
```

```
transformarCompuesto :: 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 `plegarOrigami` y su firma *(la cual debe completar con su respuesta a la parte (b))* sería la siguiente:

```
plegarOrigami :: b              -- El tipo de transformarPapel
              -> (a -> b -> b)  -- El tipo de transformarPico.
              -> a -> b -> b    -- El tipo de transformarValle.
              -> b -> b -> b    -- El tipo de transformarCompuesto.
              -> Origami a      -- El origami a plegar.
              -> b              -- El resultado del plegado.
```

Complete la definición de la función `plegarOrigami`, 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* `(Origami a)` *a considerar).*

```
plegarOrigami transPapel transPico transValle transCompuesto = plegar
    where
        plegar Papel              = transPapel
        plegar (Pico      x y) = transPico x (plegar y)
        plegar (Valle     x y) = transValle x (plegar y)
        plegar (Compuesto x y) = transCompuesto (plegar x) (plegar y)
```

d) (0.5 pts) – Usando nuestra función `plegarOrigami`, se desea implementar ahora una función `sumaOrigami`, que dado un valor de tipo `(Num a) => Origami a` calcule y devuelva la suma de todos datos almacenados en el tipo.

Complete la definición de la función `sumarOrigami`, propuesta a continuación, usando únicamente una llamada a `plegarOrigami` y definiendo las funciones de transformación necesarias. *(nótese que la función* `plegarOrigami` *recibe implícitamente el valor de tipo* `((Num a) =>Origami a)` *a considerar).*

```
sumarOrigami :: (Num a) => Origami a -> a

sumarOrigami = plegarOrigami transPapel transPico transValle transCompuesto
    where
        transPapel     = 0
        transPico      = (+)
        transValle     = (+)
        transCompuesto = (+)
```

e) (0.5 pts) – Usando nuevamente nuestra función `plegarOrigami`, se desea implementar ahora una función `aplanarOrigami`, que dado un valor de tipo `Origami a` calcule y devuelva una sola lista con todos los elementos contenidos en la estructura. En el caso de los origamis compuestos, los elementos del primer argumento deben ir antes que los del segundo.

Complete la definición de la función `aplanarOrigami`, propuesta a continuación, usando únicamente una llamada a `plegarOrigami` y definiendo las funciones de transformación necesarias. *(nótese que la función* `plegarOrigami` *recibe implícitamente el valor de tipo* `(Origami a)` *a considerar).*

```
aplanarOrigami :: Origami a -> [a]

aplanarOrigami = plegarOrigami transPapel transPico transValle transCompuesto
    where
        transPapel     = []
        transPico      = (:)
        transValle     = (:)
        transCompuesto = (++)
```

f) (0.5 pt) – Usando nuevamente nuestra función `plegarOrigami`, se desea implementar ahora una función analizarOrigami, que dado un valor de tipo `(Ord a) => Origami 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 `aplanarOrigami` estaría ordenada de menor a mayor. En el caso de un valor de tipo `Papel`, se debe devolver el valor `Nothing`. *(Pista: No es conveniente llamar explícitamente a la función* `aplanarOrigami` *para calcular el 3er elemento de la tupla.)*

Complete la definición de la función `analizarOrigami`, propuesta a continuación, usando únicamente una llamada a `plegarOrigami` y definiendo las funciones de transformación necesarias. *(nótese que la función* `plegarOrigami` *recibe implícitamente el valor de tipo* `((Ord a) =>Origami a)` *a considerar).*

```
analizarOrigami :: (Ord a) => Origami a -> Maybe (a, a, Bool)

analizarOrigami = plegarOrigami transPapel transPico transValle transCompuesto
    where
        transPapel     = Nothing
        transPico      = \x y -> merge (Just (x, x, True)) y
        transValle     = \x y -> merge (Just (x, x, True)) y
        transCompuesto = merge
            where
                merge Nothing y       = y
                merge x       Nothing = x
                merge (Just (minX, maxX, ordX)) (Just (minY, maxY, ordY)) = Just (
                     min minX minY,
                     max maxX maxY,
                     ordX && ordY && maxX < minY
                )
```

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 `plegarOrigami`, cuantas funciones debe tomar como argumento (además del valor de tipo `Gen a` que se desea plegar)?

```
Debe tomar n funciones como argumento, una por cada constructor.
```

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?

```
La función del preludio a la que correspondería sería: foldr.
```

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 imperativos. 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 imperativo 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 computos imperativos.

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

   De la definición anterior debemos notar dos cosas: el identificador `Imperativo` 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 imperativos).

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

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

   a) (0.5 pts) – ¿Por qué se tomó `(Imperativo s)` como la instancia para el monad y no simplemente `Imperativo`?

```
La clase monad está esperando un constructor de datos que reciba un argumento,
pero Imperativo recibe 2. Recordando que los constructores de datos no son sino
funciones, al pasar el primer parametro lo que resulta es un nuevo constructor
de datos que solo espera un argumento.
```

   b) (0.5 pts) – Diga las firmas para las funciones `return`, `>>=`, `>>` y `fail` para el caso especial del monad `(Imperativo s)`

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

   c) (0.5 pts) – 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 = \s -> (x, s)
```

   Notando, que `s` puede pasar al lado izquierdo de la definición y manipulando un poco la tupla, podemos encontrar una solución mucho más concisa:

```
return = (,)
```

d) (1 pt) – Complete la implementación de la función `>>=` que se da a continuación:

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

(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) (1 pt) – Demuestre que las tres leyes monádicas se cumplen para el monad `(Imperativo s)`.

a) `return x >>= f = f x`

```
        return x >>= f
= {Definición de return}
        Imperativo (\s -> (x, s)) >>= f
= {Definición de >>=}
        Imperativo $ \estadoInicial ->
            let (resultado, nuevoEstado)   = (\s -> (x, s)) estadoInicial
                (Imperativo nuevoPrograma) = f resultado
            in nuevoPrograma nuevoEstado
= {Evaluación}
        Imperativo $ \estadoInicial ->
            let (resultado, nuevoEstado)   = (x, estadoInicial)
                (Imperativo nuevoPrograma) = f resultado
            in nuevoPrograma nuevoEstado
= {Sustitución y simplificación}
        Imperativo $ \estadoInicial ->
            let (Imperativo nuevoPrograma) = f x
            in nuevoPrograma estadoInicial
= {Por su firma, debe existir un g tal que: f = (\y -> Imperativo (g y))}
        Imperativo $ \estadoInicial ->
            let (Imperativo nuevoPrograma) = (\y -> Imperativo (g y)) x
            in nuevoPrograma estadoInicial
= {Evaluación}
        Imperativo $ \estadoInicial ->
            let (Imperativo nuevoPrograma) = Imperativo (g x)
            in nuevoPrograma estadoInicial
= {Sustitución y simplificación}
        Imperativo $ \estadoInicial -> (g x) estadoInicial
= {eta-conversión: (\x -> f x) = f}
        Imperativo (g x)
= {Evaluación (en sentido contrario)}
        (\y -> Imperativo (g y)) x
= {Definición de f, introducida anteriormente}
        f x
```

b) `m >>= return = m`

```
    m >>= return
= {Por su firma, debe existir g y h tal que: m = Imperativo (\s -> (g s, h s))}
    Imperativo (\s -> (g s, h s)) >>= return
= {Definición de >>=}
    Imperativo $ \estadoInicial ->
        let (resultado, nuevoEstado)   = (\s -> (g s, h s)) estadoInicial
            (Imperativo nuevoPrograma) = return resultado
        in nuevoPrograma nuevoEstado
= {Evaluación}
    Imperativo $ \estadoInicial ->
        let (resultado, nuevoEstado)   = (g estadoInicial, h estadoInicial)
            (Imperativo nuevoPrograma) = return resultado
        in nuevoPrograma nuevoEstado
= {Sustitución y simplificación}
    Imperativo $ \estadoInicial ->
        let (Imperativo nuevoPrograma) = return (g estadoInicial)
        in nuevoPrograma (h estadoInicial)
= {Definición de return}
    Imperativo $ \estadoInicial ->
        let (Imperativo nuevoPrograma) = Imperativo (\s -> (g estadoInicial, s))
        in nuevoPrograma (h estadoInicial)
= {Sustitución y simplificación}
    Imperativo $ \estadoInicial -> (\s -> (g estadoInicial, s) (h estadoInicial)
= {Evaluación}
    Imperativo $ \estadoInicial -> (g estadoInicial, h estadoInicial)
= {Definición de m, introducida anteriormente}
    m
```

c) `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`

```
        m >>= (\x -> f x >>= g)
  = {Por su firma, debe existir mg y mh tal que: m = Imperativo (\s -> (mg s, mh s))}
        Imperativo (\s -> (mg s, mh s)) >>= (\x -> f x >>= g)
  = {Definición de >>=}
        Imperativo $ \estadoInicial ->
            let (resultado, nuevoEstado)   = (\s -> (mg s, mh s)) estadoInicial
                (Imperativo nuevoPrograma) = (\x -> f x >>= g) resultado
            in nuevoPrograma nuevoEstado
  = {Evaluación}
        Imperativo $ \estadoInicial ->
            let (resultado, nuevoEstado)   = (mg estadoInicial, mh estadoInicial)
                (Imperativo nuevoPrograma) = (\x -> f x >>= g) resultado
            in nuevoPrograma nuevoEstado
  = {Sustitución y simplificación}
        Imperativo $ \estadoInicial ->
            let (Imperativo nuevoPrograma) = (\x -> f x >>= g) (mg estadoInicial)
            in nuevoPrograma (mh estadoInicial)
  = {Evaluación}
        Imperativo $ \estadoInicial ->
            let (Imperativo nuevoPrograma) = f (mg estadoInicial) >>= g
            in nuevoPrograma (mh estadoInicial)
  = {Por su firma, debe existir un h tal que: f = (\y -> Imperativo (h y))}
        Imperativo $ \estadoInicial ->
            let (Imperativo nuevoPrograma) =
                    (\y -> Imperativo (h y)) (mg estadoInicial) >>= g
            in nuevoPrograma (mh estadoInicial)
  = {Evaluación}
        Imperativo $ \estadoInicial ->
            let (Imperativo nuevoPrograma) = Imperativo (h (mg estadoInicial)) >>= g
            in nuevoPrograma (mh estadoInicial)
```

```
= {Definición de >>=}
      Imperativo $ \estadoInicial ->
          let (Imperativo nuevoPrograma) =
             Imperativo $ \estadoInicial2 ->
                 let (resultado2, nuevoEstado2)  =
                         (h (mg estadoInicial)) estadoInicial2
                     (Imperativo nuevoPrograma2) = g resultado2
                     in nuevoPrograma2 nuevoEstado2
          in nuevoPrograma (mh estadoInicial)
= {Sustitución y simplificación}
      Imperativo $ \estadoInicial ->
          (\estadoInicial2 ->
              let (resultado2, nuevoEstado2)  = (h (mg estadoInicial)) estadoInicial2
                  (Imperativo nuevoPrograma2) = g resultado2
              in nuevoPrograma2 nuevoEstado2
          ) (mh estadoInicial)
= {Evaluación}
      Imperativo $ \estadoInicial ->
          let (resultado2, nuevoEstado2)  = (h (mg estadoInicial)) (mh estadoInicial)
              (Imperativo nuevoPrograma2) = g resultado2
          in nuevoPrograma2 nuevoEstado2
= {Sustitución y simplificación (en sentido contrario)}
      Imperativo $ \estadoInicial ->
          let (resultado2, nuevoEstado2)  =
                  let (Imperativo nuevoPrograma2) = Imperativo (h (mg estadoInicial))
                  in nuevoPrograma (mh estadoInicial)
              (Imperativo nuevoPrograma2) = g resultado2
          in nuevoPrograma2 nuevoEstado2
```

```
= {Evaluación (en sentido contrario)}
      Imperativo $ \estadoInicial ->
          let (resultado2, nuevoEstado2)  =
                  let (Imperativo nuevoPrograma2) =
                          (\y -> Imperativo (h y)) (mg estadoInicial)
                  in nuevoPrograma (mh estadoInicial)
              (Imperativo nuevoPrograma2) = g resultado2
          in nuevoPrograma2 nuevoEstado2
= {Definición de f, introducida anteriormente}
      Imperativo $ \estadoInicial ->
          let (resultado2, nuevoEstado2)  =
                  let (Imperativo nuevoPrograma2) = f (mg estadoInicial)
                  in nuevoPrograma (mh estadoInicial)
              (Imperativo nuevoPrograma2) = g resultado2
          in nuevoPrograma2 nuevoEstado2
= {Evaluación (en sentido contrario)}
      Imperativo $ \estadoInicial ->
          let (resultado2, nuevoEstado2)  =
                  (\estadoInicial2 ->
                      let (Imperativo nuevoPrograma2) = f (mg estadoInicial2)
                      in nuevoPrograma (mh estadoInicial2)) estadoInicial
              (Imperativo nuevoPrograma2) = g resultado2
          in nuevoPrograma2 nuevoEstado2
= {Definición de >>=}
      (Imperativo $ \estadoInicial2 ->
          let (Imperativo nuevoPrograma2) = f (mg estadoInicial2)
          in nuevoPrograma (mh estadoInicial2)) >>= g
= {Sustitución y simplificación (en sentido contrario)}
      (Imperativo $ \estadoInicial2 ->
          let (resultado2, nuevoPrograma2) = (mg estadoInicial2, mh estadoInicial2)
              (Imperativo nuevoPrograma2)  = f resultado2
          in nuevoPrograma nuevoPrograma2) >>= g
```

```
= {Evaluación (en sentido contrario)}
      (Imperativo $ \estadoInicial2 ->
          let (resultado2, nuevoPrograma2) = (\s -> (mg s, mh s)) estadoInicial2
              (Imperativo nuevoPrograma2)  = f resultado2
          in nuevoPrograma nuevoPrograma2) >>= g
= {Definición de >>=}
      (Imperativo (\s -> (mg s, mh s)) >>= f) >>= g
= {Definición de m, introducida anteriormente}
      (m >>= f) >>= g
```

Investigación:

(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) subs const`. 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).

```
const
```

b) (0.5 pts) – Proponga una expresión (únicamente compuesta por las funciones definidas anteriormente) cuya evaluación resulte en la misma expresión y por lo tanto nunca termine.

```
subs id id (subs id id)
```

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 = subs const ()
```

Otra posible redefinición válida, pero menos interesante podría ser `id x = const x ()`. La razón por la que la primera es preferible, es por que se corresponde directamente con la teoría de combinadores SKI (ver siguiente pregunta). Sin embargo, cualquiera de las dos (y potencialmente otras) son respuestas aceptables.

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

`El cálculo de combinadores es un modelo de cómputo (derivado y simplificado del lambda--cálculo). Está compuesto de tres combinadores: S, K e I. Dichos combinadores se comportan como las funciones subs, const e id, respectivamente. Lo interesante sobre estos combinadores, es que conforman un modelo de computo` *`Turing-Completo`*`, lo cual se traduce a que las funciones propuestas son suficientes para definir cualquier otra función computable.`