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

Tarea II: Haskell

(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.

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`.

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 :: ( ¿? )          -- El tipo de transformarPapel
              -> (a -> b -> b)   -- El tipo de transformarPico.
              -> ( ¿? )          -- El tipo de transformarValle.
              -> ( ¿? )          -- 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           = ¿?
        plegar (Pico      x y) = transPico x (plegar y)
        plegar (Valle     x y) = ¿?
        plegar (Compuesto x 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     = ¿?
        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     = ¿?
        transPico      = ¿?
        transValle     = ¿?
        transCompuesto = ¿?
```

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)?

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?

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`?

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

   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.

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

```
(Imperativo programa) >>= transformador =
    Imperativo $ \estadoInicial ->
        let (resultado, nuevoEstado)   = programa      ¿?
            (Imperativo 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) (1 pt) – Demuestre que las tres leyes monádicas se cumplen para el monad `(Imperativo s)`.

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).

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.

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.)*

d) (1 pt) – 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 `t2g<num>.pdf`, donde `<num>` es el número asignado a su equipo. Tal archivo debe ser un documento PDF con su implementación para las funciones pedidas y respuestas para las preguntas planteadas.

La tarea deberá ser entregada al prof. Carlos Perez *únicamente* a su dirección de correo electrónico oficial: (caperez@ldc.usb.ve) a más tardar el Jueves 01 de Noviembre, a las 11:59pm. VET.