Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información.
CI–3661 – Laboratorio de Lenguajes de Programación.
Enero–Marzo 2020

Tarea I: Haskell

(5 pts)

Implementación:

(2.5 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) (0.2 pts) – `miembro :: Conjunto a -> a -> Bool`
   Debe devolver la pertenencia en el conjunto proporcionado, de un elemento dado.

b) (0.2 pts) – `vacio :: Conjunto a`
   Debe devolver un conjunto vacío.

c) (0.2 pts) – `singleton :: (Eq a) => a -> Conjunto a`
   Debe devolver un conjunto que contenga únicamente al elemento proporcionado.

d) (0.2 pts) – `desdeLista :: (Eq a) => [a] -> Conjunto a`
   Debe devolver un conjunto que contenga a todos los elementos de la lista proporcionada.

e) (0.2 pts) – `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) (0.2 pts) – `union :: Conjunto a -> Conjunto a -> Conjunto a`
   Debe devolver un conjunto que contenga todos los elementos de cada conjunto proporcionado.

g) (0.2 pts) – `interseccion :: Conjunto a -> Conjunto a -> Conjunto a`
   Debe devolver un conjunto que contenga solo los elementos que estén en los dos conjuntos proporcionados.

h) (0.2 pts) – `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) (0.2 pts) – `diferenciaSimetrica :: Conjunto a -> Conjunto a -> Conjunto a`

Debe devolver un conjunto que contenga todos los elementos de cada conjunto proporcionado, pero que no contenga los que estén en ambos.

j) (0.2 pts) – `cartesiano :: Conjunto a -> Conjunto a -> Conjunto (a,a)`

Debe devolver un conjunto que contenga todas las tuplas tal que el primer elemento pertenezca al primer conjunto proporcionado, y el segundo elemento pertenezca al segundo conjunto proporcionado.

k) (0.5 pts) – `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).

Investigación:

(2.5 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) (0.5 pts) – ¿Cual es la forma normalizada para la expresión: $(\lambda x\ .\ \lambda y\ .\ x\ y\ y)\ (\lambda z\ .\ z\ O)\ L$?

b) (1 pt) – 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) (1 pt) – 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: $(\lambda x\ .\ (\lambda y\ .\ x\ y))\ y$]

Detalles de la Entrega

La entrega de la tarea consistirá de un único archivo `t1_<carné1>&<carné2>.pdf`, donde `<carné1>` y `<carné2>` son los números de carné de los integrantes de su equipo. Tal archivo debe ser un documento PDF con su implementación para las funciones pedidas y respuestas para las preguntas planteadas. Por ejemplo, si el equipo está conformado por `00-00000` y `11-11111`, entonces su entrega debe llamarse: `t1_00-00000&11-11111.pdf`.

La tarea deberá ser entregada a *ambos* profesores encargados del curso (Carlos Infante y Alexander Romero), *únicamente* a sus direcciones de correo electrónico institucionales: (13-10681@usb.ve y 13-11274@usb.ve) a más tardar el Viernes 24 de Enero, a las 11:59pm. VET.