Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información.
CI–3661 – Laboratorio de Lenguajes de Programación.
Enero–Marzo 2015

Tarea I: Haskell

(10 pts)

Implementación:

1. (2 pts) – Se dice que un número es *perfecto* si la suma de sus divisores positivos propios es igual a el mismo. Por ejemplo, el número 6 es perfecto ya que sus divisores positivos propios son 1, 2 y 3, y $1 + 2 + 3 = 6$. Los números 28 y 496 también son perfectos.

   a) (1 pt) – Implemente una función

   ```
   divisores :: Integer -> [Integer]
   ```

   que dado un número entero positivo devuelva la lista de sus divisores positivos propios en orden creciente.

   b) (1 pt) – Implemente una función

   ```
   es_perfecto :: Integer -> Bool
   ```

   que determine si un número entero positivo es perfecto.

   (Nota: Puede usar cualquier función del Preludio de `Haskell` *que crea conveniente.)*

2. (2 pts) – Un valor booleano puede ser *cierto* o *falso*. Diremos que un valor trileano puede ser *cierto*, *falso* o *desconocido*.

   a) (0.5 pts) – Implemente una estructura de datos para representar valores trileanos, que soporte impresiones en pantalla y comparaciones por igualdad.

   b) (1 pt) – Implemente la conjunción, disyunción y negación para trileanos.
   - En el caso de la conjunción, tanto *falso* como *desconocido* son absorbentes (con prioridad a *falso*).
   - En el caso de la disyunción, tanto *cierto* como *desconocido* son absorbentes (con prioridad a *cierto*).
   - En el caso de la negación, negar un *desconocido* sigue siendo *desconocido*.

   c) (0.5 pts) – Implemente un función que tome una lista de trileanos y devuelva la disyunción de todos los elementos de la misma (si la lista es vacía, la función debe evaluar a *falso*).

   (Nota: Deberá usar únicamente recursión de cola o las funciones de orden superior propuestas en el Preludio de `Haskell`*.)*

3. (3 pts) – Considere la estructura de datos `MultiConjunto`, que representa multiconjuntos (conjuntos que admiten repeticiones) potencialmente infinitos. No se tendrá una estructura que almacene explícitamente los elementos del multiconjunto, sino que se representará a través de una función que cuenta las ocurrencias de cada elemento en el multiconjunto. Elementos que no pertenezcan al multiconjunto tienen cantidad de ocurrencias 0 (cero). Por ejemplo, el multiconjunto de los números enteros pares, donde cada elemento aparece 2 veces, puede representarse como la función: `(\num -> if even num then 2 else 0)`.

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

   ```
   type MultiConjunto a = a -> Integer
   ```

   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) (0.4 pts) – `ocurrencias :: MultiConjunto a -> a -> Integer`
   Debe devolver la cantidad de ocurrencias de un elemento en el multiconjunto proporcionado.

   b) (0.3 pts) – `vacio :: MultiConjunto a`
   Debe devolver un multiconjunto vacío.

   c) (0.3 pts) – `singleton :: (Eq a) => a -> MultiConjunto a`
   Debe devolver un multiconjunto que contenga únicamente al elemento proporcionado.

   d) (0.4 pts) – `desdeLista :: (Eq a) => [a] -> MultiConjunto a`
   Debe devolver un multiconjunto que contenga a todos los elementos de la lista proporcionada.

   e) (0.4 pts) – `union :: MultiConjunto a -> MultiConjunto a -> MultiConjunto a`
   Debe devolver un multiconjunto que contenga, para cada elemento, la máxima cantidad de ocurrencias que tiene en alguno de los multiconjuntos proporcionados.

   f) (0.4 pts) – `suma :: MultiConjunto a -> MultiConjunto a -> MultiConjunto a`
   Debe devolver un multiconjunto que contenga, para cada elemento, la suma de la cantidad de ocurrencias que tiene en alguno de los multiconjuntos proporcionados.

   g) (0.4 pts) – `interseccion :: MultiConjunto a -> MultiConjunto a -> MultiConjunto a`
   Debe devolver un multiconjunto que contenga, para cada elemento, la mínima cantidad de ocurrencias que tiene en alguno de los multiconjuntos proporcionados.

   h) (0.4 pts) – `diferencia :: MultiConjunto a -> MultiConjunto a -> MultiConjunto a`
   Debe devolver un multiconjunto que contenga, para cada elemento, la resta entre la cantidad de ocurrencias que tiene en el primer multiconjunto, menos la cantidad de ocurrencias que tiene en el segundo multiconjunto (truncado a cero si dicha cantidad de hace negativa).

Investigación:

(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 corresponda una aplicación funcional, la expresión del lado izquierdo no evalúe a una λ–*abstracción*. De lo contrario, dicha λ–*abstracción* aún puede evaluarse. A continuación se presenta una semántica formal para la evaluación de λ–*expresiones*, suponiendo que **todos los identificadores presentes en las mismas son diferentes**:

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

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 los mismos sea relevante (las expresiones resultantes sean 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 será individual y consistirá de un único archivo `t1-<carné>.pdf`, donde `<carné>` es su número de carné, sin guiones ni espacios. 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 a *todos* los encargados del curso, a su direcciones de correo electrónico oficiales: (rmonascal@gmail.com, emhn@.usb.ve, manuel.gomez.ch@gmail.com, jljb1990@gmail.com & dvdalilue@gmail.com) a más tardar el Viernes 19 de Diciembre, a las 11:59pm. VET.