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 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

```
miembro x y = x y
```

Devuelve True si un elemento dado pertenece al conjunto proporcionado, Falso en caso contrario.

b) (0.2 pts) – vacio :: Conjunto a

```
vacio = \x -> False
```

Devuelve un conjunto vacío.

c) (0.2 pts) – singleton :: (Eq a) => a -> Conjunto a

```
singleton x y = x == y
```

Devuelve un conjunto que contenga únicamente al elemento proporcionado.

d) (0.2 pts) – desdeLista :: (Eq a) => [a] -> Conjunto a

```
desdeLista x y = elem y x
```

Devuelve un conjunto que contenga a todos los elementos de la lista proporcionada.

e) (0.2 pts) – complemento :: Conjunto a -> Conjunto a

```
complemento x y = not (x y)
```

Devuelve 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

```
union x y z = (x z) || (y z)
```

Devuelve un conjunto que contenga todos los elementos de cada conjunto proporcionado.

g) (0.2 pts) – interseccion :: Conjunto a -> Conjunto a -> Conjunto a

```
interseccion x y z = (x z) && (y z)
```

Devuelve 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

```
diferencia x y = interseccion x (complemento y)
```

Devuelve 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

```
diferenciaSimetrica x y = diferencia (union x y) (interseccion x y)
```

Devuelve 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)

```
cartesiano x y (m,n) = miembro x m && miembro y n
```

Devuelve 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

```
transformar x y z = y (x z)
```

Devuelve 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:

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 subexpresión 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):

(A) eval(x) = x.
(B) eval(λx . E) = λx.eval(E)
(C) eval(x F) = x eval(F)
(D) eval((λx . E) F) = eval(E) [x := eval(F)]
(E) eval((E F) G) = eval(eval(E F) eval(G))
Tomando esta definición en cuenta, conteste las siguientes preguntas:

a) (0.5 pts) – ¿Cual es la forma normalizada para la expresión: (λx . λy . x y y) (λz . z O) L?

```
(A) eval(x) = x.
(B) eval(λx . E) = λx.eval(E)
(C) eval(x F ) = x eval(F )
(D) eval((λx . E) F ) = eval(E) [x := eval(F)]
(E) eval((E F ) G) = eval(eval(E F ) eval(G))

eval( ( (λx . ( λy . ( x y y ) ) ) ( λz . ( z O ) ) ) L)
      Aplico (E)
eval( eval( ( λx (λy ( x y y ) ) ) ( λz ( z O ) ) ) eval( L ) )
      Aplico (D)
eval( eval ( λy ( x y y ) ) [x := eval( λz ( z O ) ) ] ) eval( L )
      Aplico (D)
eval( eval ( eval ( λy ( eval( λz ( z O ) ) y y ) [ y := eval ( L ) ] ) ) )
      Aplico (D)
eval( eval ( eval( eval( z O ) [ z := eval ( eval ( L ) ) ] eval ( L ) ) ) )
      Aplico (A)
eval ( eval ( eval ( eval ( eval ( L O ) eval ( L ) ) ) ) )
      Aplico (A)
eval ( eval ( eval ( eval ( L O L ) ) ) )
      Aplico (A)
eval ( eval ( eval ( L O L ) ) )
      Aplico (A)
eval ( eval ( L O L ) )
      Aplico (A)
eval ( L O L )
      Aplico (A)
L O L
      Ya no se puede aplicar ninguna regla,esta es la forma normalizada
```

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.
Si, efectivamente el orden en que se evalúan las expresiones es relevante. Tomemos:

```
E = (λx (-1))     F = ((λy (y y)) (λy(y y)))
```

```
(λx (-1)) ((λy (y y)) (λy(y y)))
```
Si evaluamos primero E, es decir, de afuera hacia adentro obtenemos:
```
-1
```
Esto es debido a que E es una función que, sin importar cual sea el argumento, evalua -1.
```
(λx (-1)) ((λy (y y) λy(y y)))
(λx (-1)) ((λy (y y) λy(y y)))
```
En cambio si evaluamos primero F nos damos cuenta de que es una recursión infinita, por lo cual nunca vamos a terminar de calcular el argumento para E.

c) (1 pt) – Considere una evaluación para una λ–expresión de la forma ((λx . E) F). ¿Qué cambios haría a la semántica formal de la función eval para este caso, si se permitiese identificadores repetidos? [Considere, como ejemplo, lo que ocurre al evaluar la siguiente expresión: (λx . (λy . x y)) y]
Al evaluar la expresion dada, la corrida luce de esta manera:
```
eval((λx . (λy . x y)) y)
(D)
eval(λy . x y) [x := eval( y )]
(SUSTITUCION)
eval(λy . eval(y) y)
(A)
eval(λy . y y)
(B)
λy . eval(y y)
(A)
λy . y y
```

Ahora, bien cambiemos la expresión por una equivalente: (λx . (λy . x y)) z

```
eval((λx . (λy . (x y))) z)
(D)
eval(λy . (x y)) [x := eval( z )]
(SUSTITUCION)
eval(λy . (eval(z) y))
(A)
eval(λy . (z y))
(B)
λy . eval(z y)
(A)
λy . z y
```

Sin embargo, en el primer resultado ambas variables están ligadas. Mientras que en la segunda solo una variable está ligada. Esto es un problema dado que dos expresiones equivalentes no produjeron resultados equivalentes tras la misma corrida. Más aún, el primero es incorrecto, pues liga una variable que era independiente. Para corregir esto, se propone agregar la condición a la regla (D) de que si no ocurre libre una variable F en E dicha variable se debe renombrar.