Programación Declarativa 2020-2
Facultad de Ciencias UNAM
Tarea 6: Release the Monad!

Favio E. Miranda Perea Javier Enríquez Mendoza

Fecha de entrega: 8 de junio de 2020

Todos los ejercicios de esta tarea deben resolverse usando mónadas

1. *(1 pt)* Las máquinas del tiempo tienen un defecto de fábrica que causa un mal funcionamiento, por ejemplo, imagina que tu máquina salta de forma aleatoria entre:

   - Un año hacia el pasado
   - Tres años al futuro
   - Cinco años en el futuro

   Entonces si quieres viajar cuatro años en el futuro, necesitas un salto de cinco años hacia el futuro y luego retroceder un año ¡Qué molesto!

   El problema es que cada que viajas en el tiempo solo puedes contar el número de saltos dados y al final del viaje no sabes en que año te encuentras. Diseña una función que dado el número de saltos y el año del que partiste te dé todos los posibles años en los que podrías encontrarte después del viaje.

   ```
   timeTravel :: Int -> Int -> [Int]
   ```

   El primer parámetro es el número de saltos, mientras que el segundo es el año en el que comienza el viaje. Para esta función utiliza la instancia de la clase `Monad` para listas. Define primero la función `timeTravel` usando únicamente los operadores mónadicos y después escribe la función `timeTravelD` que haga lo mismo pero usando notación `do`.

2. *(1.5 pts)* Un *Quine* es un metaprograma que produce su código fuente como única salida, es decir, un programa que se imprime a sí mismo. Solo hay una regla:

   Abrir el archivo fuente del programa e imprimir el contenido se considera hacer trampa.

   Algunos ejemplos de *Quine* en diferentes lenguajes son:

   Scheme

```
((lambda (x)
    (list x (list (quote quote) x)))
 (quote
     (lambda (x)
      (list x (list (quote quote) x)))))
```

Python

```
a='a=%s;print a%%`a`;print a%`a`
```

JavaScript

```
unescape(q="unescape(q=%22*%22).replace('*',q)").replace('*',q)
```

Brainfuck

```
->+>+++>>+>+++>+>+++>>+>++>>>+>+>+>++>+>>>>+++>+>>++>+>+++>+
+>++>>+>>>+>++>++>+>>>>+++>+>>>>++>++>>>>+>>++>+>+++>>>++>>++
++++>>+>>>++>+>>>>>+++>>+++++>>+>+++>>>++>>+++>>+>>++>+>+++>>>+
+>>++++++++++++++>>+>>+++>+>+++>+>+++>>>++>>+++++>>+>>++>+>>>>+
++>>+++++>>>>++>>>>+>+>++>>+++>+>>>>+++>+>>>>+++>+>>>>+++>>+
+>++>+>+++>+>++>++>>>>>>>++>+>+++>>>>>+++>>>++>+>+++>+>+>++>>
>>>>++>>>>+>>>>++>+>>>>>+++>+>>>>+>>++>+>++++++++++++++++++>>>>+
>+>>>+>>>++>+>+++>>>++>>>+++++++>>+>>++>+>>>>+++>>++++++>>>+>
++>>+++>+>+>++>+>+++>>>>>+++>>>>+>+>>++>+>+++>>>++>>>+++++++>
>+>>>++>+>>>>+++>>++++>>+>+++>>>>>>>++>+>+++>>>+>++>>>>+>+>++>+
>>>>+++>>+++>>>>+[[->>+<<]]<+]+++++[->+++++++++<]>.[+]>>[<<++
+++++[->+++++++++<]>-.------------------->-[-<.<+>>]<[+]<+>>
>]<<<[-[-[-[>>+<++++++[->+++++<]]>+++++++++++++<]>+++<]++++
++[->+++++++<]>+<<<-[->>>++<<<]>[->>.<<]<<]
```

Cálculo Lambda

```
(λx.xx)(λx.xx)
```

Este ejercicio consiste en escribir un *Quine* en Haskell.

3. *(1.5 pts)* La conjetura de Goldbach es un problema abierto, fue enunciada por Christian Goldbach en una carta para Euler en 1742 y afirma lo siguiente:

   Todo número entero mayor que 5 se puede escribir como suma de tres números primos.[1]

   Definir la función `goldbach` que dado un entero n, si $n > 5$ devuelve una lista con ternas `(i,j,k)` tales que `i + j + k` $=n$ y todos los elementos de la terna son números primos, en otro caso se regresa la lista vacía.

[1] Esta versión es la original que resulta equivalente a la que es más conocida: *Todo número par mayor que 2 puede escribirse como suma de dos números primos.*

```
goldbach :: Int -> [(Int,Int,Int)]
```

Esta función debe implementarse haciendo uso únicamente de los operadores para mónadas. Definir también `golbachD` con el mismo comportamiento pero usando notación `do`.

Hint: Recuerda que las listas por comprensión son azúcar sintáctica para la notación `do` que a su vez es azúcar sintáctica de los operadores monádicos.

4. *(2 pts)* Se desea implementar otra versión del evaluador de expresiones visto en clase, para que sea capaz de regresar el resultado, si la operación fue exitosa o un error **descriptivo**, si ésta fallo. Para esto se va a extender el tipo `Expr` como sigue:

```
data Expr = Num Int | Var Int | Div Expr Expr
```

usando el constructor `Var` para modelar variables, en donde el entero con el que se construyen será su identificador. Entonces la evaluación ahora dependerá de un ambiente de variables que será modelado con una lista de pares como se muestra enseguida

```
type Value = Int
type Env = [(Int,Value)]
```

la primera entrada del par corresponde al identificador de la variable mientras que el segundo es el valor de ésta (solo se admiten valores de tipo entero).

Para modelar las excepciones se define el tipo `Exception`

```
data Exception = DivisionPorCero | VariableNoDefinida | MalaSuerte
```

Y con este se modela el tipo `ExprErr` que modela los cómputos que podrían lanzar excepciones.

```
data ExprErr a = Raise Exception | Return a
```

los constructores `DivisionPorCero` y `VariableNoDefinida` modelan los errores descritos por su nombre mientras que `MalaSuerte` es un error que ocurrirá siempre que durante el cómputo aparezca el número 13 ya sea como constante, valor de una variable o como resultado de un cómputo. Si existiera mas de un error en la evaluación solo se reportará el primero.

Para poder construir este evaluador es necesario:

(a) Dar una instancia del tipo `ExprErr` de la clase `Monad`, recordando que se deben cumplir las leyes mónadicas (No es necesario dar la prueba)

(b) Definir una función monádica `evalEx` que efectúa la evaluación reportando los errores. Definir el tipo de esta función es parte del ejercicio pero recuerda que se necesita de un ambiente para la evaluación de variables.

5. *(2 pts)* La siguiente función en Haskell calcula el máximo común divisor de dos números, usando el algoritmo de Euclides.

```
gcd :: Int -> Int -> Int
gcd a b
  | a < b           = gcd b a
  | a `mod` b == 0 = b
  | otherwise       = gcd b (a `mod` b)
```

Queremos definir ahora una función que calcule el máximo común divisor de dos números, pero que regrese el registro del proceso que se hizo para llegar al resultado. En lenguajes imperativos como Python, basta con con imprimir el paso realizado en cada iteración, pero esto no es tan sencillo en Haskell. Para lograrlo definimos el tipo `Log` que contiene el valor final así como una lista de los pasos necesarios para calcularlo, este tipo se define como sigue:

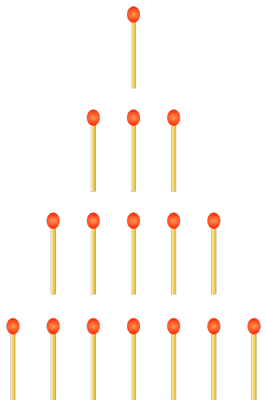
```
data Log a = Log {Value :: a, Logs :: [String] }
```

Cada paso se representa como una cadena que describe la operación realizada en este punto.

(a) Crea una instancia de la clase `Monad` para el tipo `Log`.
En donde `return` regresa un valor dentro de un `Log` sin ningún paso, es decir, con la lista vacía.
Mientras que el operador `(log >>= f)`, aplica la función `f` al valor ya almacenado en `log`, lo que regresa un nuevo `Log` al cual se le concatenan los pasos que ya se habían realizado.
Hint: Revisa los tipos de los métodos de la clase `Monad`.

(b) Implementa la función `gcdlog :: Int -> Int -> Log Int` que calcula el máximo común divisor entre dos números con el registro de los pasos realizados. Para está función es necesario utilizar la instancia creada en el inciso anterior.

6. *(2 pts)* Para el juego de Nim se utiliza el siguiente tablero inicial:



Las reglas del juego son:

- Hay dos jugadores. Que juegan por turnos alternadamente.
- En su turno el jugador puede retirar el número de cerillos que desee, con la única condición de que estos estén en la misma fila.
- El jugador está obligado a retirar al menos un cerillo en su turno.
- El jugador que retira el último cerillo pierde.

Implementar en Haskell el juego de Nim, mediante un programa interactivo usando `IO`, en el que cada iteración corresponde a un turno del juego, al inicio de cada turno se debe imprimir el tablero actual e indicar a que jugador le corresponde el turno, el turno termina cuando el jugador indica cuántos cerillos quiere eliminar y de qué fila, al finalizar el juego el programa debe informar quién fue el ganador.

Este es un ejemplo de un turno del juego:

```
Turno del jugador 1
1:        *
2:      * * *
3:    * * * * *
4: * * * * * * *

> 2
> 4

Turno del jugador 2
1:        *
2:      * * *
3:    * * * * *
4: * * * * *

>
```

Esta representación es solo un ejemplo, se puede representar el tablero de otra forma, mientras sea comprensible.

El programa debe ser robusto, es decir reaccionar adecuadamente a entradas no esperadas, o entradas no válidas como intentar eliminar mas cerillos de los que tiene una fila, y no cambiar de turno hasta que se tenga una entrada válida del jugador.

Entrega

- La tarea puede entregarse de forma individual o en parejas.
- Se tomará en cuenta tanto el estilo de programación como la complejidad de las soluciones para la calificación de la sección práctica.
- La tarea se entrega a través de Slack:
  - Todos los archivos deben tener al principio como comentario los nombres de los alumnos.
  - Si la tarea se realizo en parejas crear un canal privado con el ayudante y los miembros del equipo para la entrega.
  - Si se realizo de forma individual enviarla en un mensaje directo al ayudante.
  - **No** deben comprimir los archivos, en caso de tener mas de uno enviarlos por separado.