Programación Funcional

Trabajo Práctico Nro. 9

Temas: Functores y monadas.

1. Considere la siguiente clase que captura la interfaz común a paquetes (`Package`) contenedores de datos:

   ```
   class Package p where
     handle :: (a -> b) -> p a -> p b
   ```

   Que indica que todo paquete `p` que contiene datos de tipo `a` puede ser manipulado usando la función `handle` y una función de tipo `a -> b`. Esto genera un nuevo paquete de igual estructura `p` pero que contiene datos de tipo `b`.

   De una implementación de la clase `Package` usando el keyword `instance` para los tipos:

   a) Maybe (en Haskell `Maybe`)

   b) Listas (en Haskell `[]`)

   c) Funciones de `a -> b` (En Haskell use la notación `((->) a)` para referirse a las funciones que toman argumentos de tipo `a`).

2. Considere la siguiente clase con la interfaz común a cajas (`Box`) contenedoras de datos:

   ```
   class Box m where
     link :: (a -> m b) -> m a -> m b
     pack :: a -> m a
   ```

   Que indica que toda caja `m` que contiene datos de tipo `a` puede ser utilizada usando dos operaciones: `link` y `pack`. Pack dado un elemento de tipo `a` simplemente construye una caja que contiene ese elemento. Link toma una función de tipo `a -> m b`, es decir, dado un elemento de tipo `a` construye una caja que contiene elementos de tipo `b`; y dada una caja con elementos de tipo `a` aplica la función a cada elemento almacenado. Finalmente construye una única caja con los valores resultantes al mismo nivel.

   De una implementación de la clase `Box` usando el keyword `instance` para los tipos:

   a) Maybe

   b) Listas

   c) Funciones de `a -> b`

Luego escriba las siguientes funciones utilizando la interfaz `Box` y reutilizando las operaciones ya implementadas de ser necesario:

a) `handle' :: Box m => (a -> b) -> m a -> m b`
Que demuestra que toda caja es un paquete, es decir, que la interfaz `Box` es más general que la interfaz `Package`.

b) `unite :: Box m => m (m a) -> m a`
Que une una caja de cajas en una sola caja donde los elementos contenidos están al mismo nivel.

c) `employ :: Box m => m (a -> b) -> m a -> m b`
Que dada una caja que contiene funciones y una caja con elementos construye una caja con los resultados de emplear cada función para procesar cada elemento.

d) `associate :: Box m => (a -> m b) -> [a] -> m [b]`
Que dada una función que construye cajas y una lista de elementos retorna una caja de listas.

e) `succession :: Box m => [m a] -> m [a]`
Que dada una lista de cajas construye una única caja con la lista de elementos resultante de recolectar secuencialmente los elementos en las cajas.

3. Observe que la interfaz `Pacakge` es equivalente a la interfaz `Functor` y que la interfaz `Box` es equivalente a la interfaz `Monad` definidas en la librería estándar de Haskell. Investigue cuáles son los nombres que se le dan en la librería estándar a las funciones implementadas en los puntos anteriores. Liste también las propiedades que deben cumplir las funciones de la interfaz para efectivamente poder considerar a un tipo de datos una mónada.

4. Considere la interfaz `Appendable` que captura estructuras que pueden ser concatenadas.

```
class Appendable a where
  empty  :: a
  append :: a -> a -> a
```

Para el tipo de las tuplas cuyo primer componente es `Appendable` de una implementación de las interfaces `Functor` y `Monad` (en haskell `Appendabe c => ((,) c)`). Esta es conocida como la mónada *Writer* y sirve, por ejemplo, para modelar cómputo con un log. Es decir, cada operación agrega mensajes al log, pero no los lee ni los borra.

5. Considere el siguiente tipo que contiene una función:

```
newtype ST s a = ST (s -> (s,a))
```

De una implementación de las interfaces `Functor` y `Monad` para el tipo `ST s`. La intuición detrás de la función contenida en un `ST` es que dado un estado `s` la función retorna un valor `a` y al mismo tiempo "actualiza" el estado. Esta es conocida como la mónada *State*.

6. Para los siguientes extractos de código identifique la mónada en uso, el tipo de las expresiones y traduzca de notación `do` a la notación con la función de *bind* (`>>=`) explícita.

   a)
   ```
   pred3 x = do n <- pred x
                m <- pred n
                o <- pred m
                return o
     where pred Z     = Nothing
           pred (S x) = Just x
   ```

   b)
   ```
   perms xs = if null xs then
                return []
              else do {
                x   <- xs;
                xs' <- perms (rem x xs);
                return $ x : xs';
              }
            where rem y (x:xs) | y == x    = xs
                               | otherwise =  x : rem y xs
   ```

   c)
   ```
   eciwt = do {x <- id; y <- (.x); return y}
   ```

7. El problema de las n-reinas consta en encontrar todas las formas posibles de colocar, en un tablero de ajedrez de $n \times n$, n reinas sin que se amenacen. Las reinas amenazan a cualquier pieza en su misma fila, columna o diagonal. Use la mónada que modela el cómputo no-determinístico (la lista) para escribir un algoritmo de *backtracking* que resuelva este problema. Un algoritmo de *backtracking* considera todas las secuencias de alternativas posibles siempre y cuando sus prefijos satisfagan una propiedad dada, en caso contrario evitan seguir analizando las consecuencias de tal alternativa. Note que en el caso de las n-reinas no es necesario continuar el análisis de un tablero donde ya hay reinas amenazándose, por lo que todo prefijo de colocación de piezas debe cumplir la propiedad de que no hay reinas amenazadas.

   Considere definir:

   - `type Coord = (Int, Int)` y `type Board = Coord -> Bool`
   - `nqueens :: Int -> [Board]`
     La función que dado el n retorna la lista con todos los tableros posibles de $n \times n$ en donde hay n reinas que no se amenazan.
   - `addQueen :: Board -> Coord -> Board`
     La función que agrega a un tablero una reina en una coordenada dada.
   - `threatened :: Board -> Int -> Coord -> Bool`
     La función que determina si en un tablero (de $n \times n$) una coordenada dada está amenazada por alguna pieza.

- Cualquier función auxiliar que necesite, siempre y cuando resuelva el problema principal usando mónadas.

8. El problema llamado *Knight's tour* consta en obtener una secuencia de movimientos para un caballo en un tablero de ajedrez comenzando en una esquina, de forma tal que se visiten todas los casilleros una única vez. Use la mónada que modela el cómputo no-determinístico (la lista) para escribir un algoritmo de *backtracking* que resuelva este problema.

   Considere utilizar:

   - Los tipos `Coord` y `Board` definidos en el ejercicio anterior.
   - `tour :: Int -> [Coord]`
     La función que dado un tamaño de tablero retorna la lista con una secuencia de coordenadas que llevan a recorrer todos los casilleros del tablero una única vez utilizando el movimiento del caballo (o vacía si tal secuencia no existe para el tamaño de tablero dado).
   - `visit :: Board -> Coord -> Board`
     La función que marca un casillero como visitado.
   - `moves :: Int -> Board -> Coord -> [Coord]`
     La función que a partir del tamaño del tablero, el tablero y una coordenada, retorna una lista de coordenadas no visitadas que el caballo puede alcanzar en un movimiento desde la coordenada dada.
   - Cualquier función auxiliar que necesite, siempre y cuando resuelva el problema principal usando mónadas.

   Agregue a su implementación el criterio de *Warnsdorf* que prioriza el análisis de los movimientos que llevan a casillas no visitadas con la menor cantidad de posibles movidas. Es decir, los movimientos válidos para el caballo a partir del casillero actual deben ordenarse según este criterio y continuar el análisis en orden.

   Se recomienda usar:

   - `warnsdorf :: Int -> Board -> Coord -> Coord -> Ordering`
     La función que compara dos coordenadas de un tablero utilizando el criterio de *Warnsdorf*. Si la primer coordenada tiene menos movidas válidas que la segunda retorna el valor `LT`, en caso que sean iguales retorna `EQ` y en caso de que sea mayor retorna `GT` (el tipo `Ordering` está definido en el *Prelude*).
   - `sortBy :: (a -> a -> Ordering) -> [a] -> [a]`
     Función del paquete `Data.List` que dado un criterio de comparación y una lista ordena la lista siguiendo el criterio.

   Compare los tiempos de ejecución de ambas implementaciones para un tablero de 8×8.

9. `Imp` es un lenguaje imperativo simple que opera con valores enteros. Implemente un interprete utilizando una mónada que tenga en cuenta el estado de la memoria y lo actualice en caso de ejecutar una operación con efecto. Tenga en cuenta las siguientes definiciones:

```
type Variable = String -- tipo para los identificadores de variables
type Value = Int       -- tipo para los valores procesados
type Store = Variable -> Value -- tipo para el estado de la memoria
type BinOp = Value -> Value -> Value -- tipo de los operadores

data Expression =  -- tipo de las Expressionesiones
     Cte Int                         | -- constante entera
     Var Variable                    | -- variable (lectura de memoria)
     Bin BinOp Expression Expression   -- aplicación de un operador binario

data Statement =  -- tipo de las sentencias
     Assign Variable Expression        | -- asignación de una variable
     Block [Statement]                 | -- bloque de sentencias
     If Expression Statement Statement | -- construcción condicional
     While Expression Statement          -- construcción de ciclo

type Program = Statement  -- tipo de los programas IMP

-- Tipo del estado de un interprete de un programa IMP
newtype Interpreter v = Imp { step :: Store -> (Store, v) }

-- Función que lee el valor de una variable de la memoria
load :: Variable -> Interpreter Value
load x = Imp $ \r -> (r, r x)

-- Función que escribe un valor de una variable en memoria
write :: Variable -> Value -> Interpreter ()
write x v = Imp $ \r -> (\y -> if y == x then v else r y, ())
```

Defina:

a) La instancia de la clase `Monad` para el tipo `Interpreter`.

b) `eval :: Expression -> Interpreter Value`
Función que evalúa una expresión.

c) `exec :: Statement -> Interpreter ()`
Función que ejecuta una sentencia.

d) `start :: Program -> Value`
Función que ejecuta un programa desde un inicio "limpio" y retorna como resultado el contenido de la variable llamada `"result"`. **Ayuda:** considere crear un `Store` inicial donde toda variable está asociada con el valor por default `0`.

10. Considere la siguiente clase que engloba las mónadas con la capacidad adicional de poder combinarse y que tienen un elemento neutro para esta combinación.

```
class (Monad m) => MonadPlus m where
  mzero :: m a               -- elemento neutro
  mplus :: m a -> m a -> m a -- opercaion de combinacion
```

Las siguientes definiciones dan las bases para construir *Parsers* de `Strings`. Un parser consume caracteres de un string y genera múltiples resultados asociados con distintas reglas de parseo (cero o más resultados). Decimos que un parser es exitoso y acepta una cadena cuando retorna al menos un resultado. Observe que la aplicación de un parser también retorna la cola del string sobre la cual no se consumieron caracteres.

```
-- Tipo de un parser, contiene una funcion que dado un string retorna
-- cero o mas tuplas de resultados y colas de caracteres sin consumir
newtype Parser a = Parser (String -> [(a,String)])

-- Funcion que aplica un parser a un String
parse :: Parser a -> String -> [(a,String)]
parse (Parser p) s = p s

-- Parser que consume un caracter y lo retorna como resultado
item :: Parser Char
item = Parser $ \s -> case s of {(c:s') -> [(c,s')]; [] -> []}

-- Parser que es exitoso solo cuando el input es vacio
end :: Parser ()
end = Parser $ \s -> if null s then [((),[])] else []
```

Defina:

- La instancia de la clase `Monad` para el tipo `Parser`.
- La instancia de la clase `MonadPlus` para el tipo `Parser`.

Utilizando las operaciones anteriores y **sin utilizar** la estructura de representación de un `Parser` defina:

a) `fails :: Parser a`
Un parser que falla sin consumir input (fallar se modela retornando una lista vacía).

b) `chain :: Parser a -> Parser b -> Parser (a,b)`
Que retorna el parser resultante de aplicar dos parsers en secuencia.

c) `choice :: Parser a -> Parser a -> Parser a`
Que retorna el parser resultante de aplicar otros dos en orden sobre el **mismo input**.

d) `satisfy :: (Char -> Bool) -> Parser Char`
Que consume un carácter si satisface una propiedad dada o falla en caso contrario.

e) `terminal :: Char -> Parser Char`
Un parser que consume un carácter dado.

f) `apply :: (a -> b) -> Parser a -> Parser b`
Un parser que aplica una función al resultado de otro parser.

g) `many :: Parser a -> Parser [a]`
Un parser que aplica otro cero o más veces hasta que deja de ser exitoso (siempre se considera exitoso aunque no se logre aplicar el parser dado ni una vez).

h) `many1 :: Parser a -> Parser [a]`
Un parser que aplica otro una o más veces hasta que deja de ser exitoso (sólo falla si no logra aplicar exitosamente el parser dado al menos una vez).

Teniendo la función que a partir de una cadena de caracteres numéricos retorna el entero representado:

```
readInt :: String -> Int
readInt s = read s :: Int
```

Defina el parser de expresiones numéricas con suma y producto, teniendo en cuenta que el parser además de leer la expresión tiene que evaluarla. Para ello considere definir las funciones:

i) `num :: Parser Int`
Que parsea un número.

j) `par :: Parser Int`
Que parsea una expresión numérica entre paréntesis.

k) `add :: Parser Int`
Que parsea una expresión numérica compuesta por la suma de dos expresiones.

l) `mul :: Parser Int`
Que parsea una expresión numérica compuesta por el producto de dos expresiones.

m) `pex :: Parser Int`
Que parsea cualquier expresión numérica dejando en el output del parser el resultado de evaluarla. Es exitosa sólo cuando consume el input dado por completo.

Ayuda: Para mantener la asociatividad de las operaciones considere parsear el operando a la izquierda de la suma con el parser del producto. De la misma forma considere parsear el operando a la izquierda de la multiplicación como una expresión unaria (o bien un número o bien una expresión entre paréntesis). De esta forma además se garantiza evitar llamadas recursivas a izquierda que harían entrar al parser en una recursión infinita.