Práctica 7

Programación Funcional, UNQ

Mónadas

Aclaraciones:

- *Los ejercicios fueron pensados para ser resueltos en el orden en que son presentados. No se saltee ejercicios sin consultar antes a un docente.*
- *Recuerde que puede aprovechar en todo momento las funciones que ha definido, tanto las de esta misma práctica como las de prácticas anteriores.*
- *Pruebe todas sus implementaciones, al menos en una consola interactiva.*
- *Es sumamente aconsejable resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en clase, dado que los exámenes de la materia evaluarán principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*
- *No dude en manifestar observaciones y críticas sobre los ejercicios de esta práctica, que con gusto serán recibidas por los docentes.*
- *Los ejercicios del anexo pueden obviarse, pero recuerde que aportan una compresión más profunda sobre los temas que aborda esta práctica. Considere resolverlos si se encuentra practicando para una instancia de evaluación y ya resolvió todos los anteriores.*

1. Generic

Considere la siguiente definición de la clase `Monad`:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Y defina las siguientes operaciones usando la interfaz (i.e., independiente de una implementación particular):

a) `liftM :: Monad m => (a -> b) -> m a -> m b`, que aplica una función sobre la mónada retornando el resultado en la mónada.

b) `join :: Monad m => m (m a) -> m a`, que aplana una mónada anidad en una sola mónada.

c) `ap :: Monad m => m (a -> b) -> m a -> m b`, que aplica una mónada de tipo función a una mónada con el tipo del parámetro.

2. Maybe

Instancie la clase `Monad` para el tipo `Maybe` · (dando su bind y return). Luego traduzca el siguiente fragmento de código eliminando la sintaxis especial `do`.

```
pred3 x =
  do n <- pred x
     m <- pred n
     o <- pred m
     return o
```

```
  where pred Z     = Nothing
        pred (S x) = Just x
```

Y responda cuál es el resultado para los siguientes casos:

a) `pred3 (S Z)`

b) `pred3 (S (S Z))`

c) `pred3 (S (S (S Z)))`

3. List

Instancie la clase `Monad` para el tipo `[·]` (dando su bind y return). Luego traduzca el siguiente fragmento de código eliminando la sintaxis especial `do`.

```
perms xs =
  if null xs then
    return []
  else do
    x   <- xs
    xs' <- perms (xs // [x])
    return (x:xs')
```

Y responda cuál es el resultado para los siguientes casos:

a) `perms [1]`

b) `perms [1, 2]`

c) `perms [1, 2, 3]`

4. Reader

Instancie la clase `Monad` para el tipo de las funciones `k ->` · (dando su bind y return). Luego traduzca el siguiente fragmento de código eliminando la sintaxis especial `do`.

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

Y responda cuál es el resultado para los siguientes casos:

a) `twice (+1) 0`

b) `twice (*2) 1`

5. Writer

Considere la interfaz `Appendable` que captura estructuras que pueden ser concatenadas (i.e., admiten una función `append`).

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

Instancie la clase `Monad` para el tipo de las tuplas cuyo primer componente es `Appendable` (dando su bind y return). Tenga en cuenta que en Haskell el tipo se escribe "currificado" (i.e., `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.

Como ejemplo, defina la siguiente función:

a) `greaterThan :: Int -> [Int] -> (String, [Int])`, que dado un número se queda con los elementos que son mayores a éste en una lista, informando qué elementos son agregados al resultado.

6. State

Considere el siguiente tipo que contiene una función:

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

Instancie la clase `Monad` para el tipo `State s` · (dando su bind y return). 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* y sirve para modelar cómputo con estado.

Por ejemplo, dada la siguiente definición del tipo `Stack`:

```
type Stack = [Int]
```

Implemente las siguientes operaciones:

a) `pop :: State Stack Int`, que remueve y retorna (dentro de la mónada) el elemento en el top del Stack, asumiendo que el stack no está vacío.

b) `push :: Int -> State Stack ()`, que agrega un elemento dado como tope del Stack.

c) `popN :: Int -> State Stack [Int]`, que **sin utilizar el constructor ST** (i.e., utilizando las operaciones de la clase Monad más pop y push), remueve n elementos y retorna (dentro de la mónada) la lista de elementos removidos.

7. MonadPlus

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
```

Instancie la clase para las siguientes mónadas:

a) `Maybe` ·

b) `[·]`

c) `State s` ·

8. Ejercicio integrador*

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.