# CI4251 - Programación Funcional Avanzada
# Tarea 2

Ernesto Hernández-Novich
86-17791
<emhn@usb.ve>

Mayo 18, 2013

## Arboles de Expresiones y su Evaluación

```
import qualified Data.Map as DM
```

Considere el siguiente TAD diseñado para representar árboles abstractos conteniendo expresiones aritméticas sobre números enteros y variables:

```
data Exp = Const Int              -- constante manifiesta
         | Var String             -- nombre de variable
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Div Exp Exp
         deriving (Eq,Show)
```

Queremos evaluar las expresiones representadas por esos árboles en el contexto de un ambiente de referencia *inmutable* que asocia valores a variables apoyándose en `Data.Map`

```
env = DM.fromList [
                    ("foo",42), ("bar",69),
                    ("baz",27), ("qux",0)
                  ]
```

La intención del tipo de datos es construir expresiones como [1]

```
ex1 = (Const 21)
ex2 = (Var "foo")
ex3 = (Mul (Const 2) ex1)
ex4 = (Add (Mul ex2 (Const 5)) (ex3))
ex5 = (Add (Div (Const 42) (Const 2))
           (Mul (Const 3) (Const 7))
      )
ex6 = (Div (Var "bar")
           (Sub (Add (Const 2) (Const 3)) (Const 5)))
ex7 = (Div (Add (Const 6) (Const 7)) (Const 13))
```

Evaluar esas expresiones de manera *pura* es simple

```
eval (Const i) = i
eval (Var n)   = fromJust (DM.lookup n env)
eval (Add l r) = (eval l) + (eval r)
eval (Sub l r) = (eval l) - (eval r)
eval (Mul l r) = (eval l) * (eval r)
eval (Div l r) = (eval l) `div` (eval r)
```

[1] La indentación en las expresiones es simplemente para facilitar su comprensión – es importante que Ud. construya expresiones de prueba más complejas para ejercitar correctamente su solución. Note que la expresión `ex6` contiene una división por cero

## Evaluador monádico con ambiente fijo (3 puntos)

Se desea implantar el mismo evaluador, pero usando un monad que acompañe el cómputo con el ambiente de evaluación.

El evaluador debe estar encapsulado en una función tal que pueda evaluarse en GHCi directamente, i.e.

```
ghci> evalwithenv env ex4
Result: 252
```

Para conseguirlo, es necesario que Ud. provea:

- Un combinador monádico aprovechando `Control.Monad.Reader` que complete la evaluación de las expresiones acarreando el ambiente de evaluación consigo

```
evalRD :: {- Firma para aprovechar Control.Monad.Reader -}
```

- La función principal de evaluación que aprovecha `evalRD` y las funciones monádicas de `Control.Monad.Reader` para efectuar la evaluación y mostrar los resultados

```
evalwithenv :: DM.Map String Int -> Exp -> IO ()
```

Si la expresión a evaluar contiene una división por cero, o intenta utilizar un símbolo que no está definido en la tabla de símbolos, es suficiente que emita un mensaje de error usando `error`.

```
ghci> evalwithenv env ex6
Result: *** Exception: divide by zero
ghci> evalwithenv env (Var "meh")
evalwithenv (Var "meh")
Result: *** Exception: Var meh not found
```

## Evaluador con cálculo de costos (3 puntos)

Se desea implantar un evaluador de expresiones que sea capaz de contabilizar la cantidad de sumas, restas, multiplicaciones, divisiones y consultas a la tabla de símbolos que se efectúan durante el cómputo.

El evaluador debe estar encapsulado en una función tal que pueda evaluarse en GHCi directamente, i.e.

```
ghci> evalwithstats ex4
Result: 252
EvalState {adds = 1, subs = 0, muls = 2, divs = 0, vars = 1}
```

Para conseguirlo, es necesario que Ud. provea:

- El tipo de datos que almacena las estadísticas

```
data EvalState = ...
```

- Un combinador monádico aprovechando `Control.Monad.State` que complete la evaluación de las expresiones acarreando las estadísticas

```
evalST :: {- Firma para aprovechar Control.Monad.State -}
```

- La función principal de evaluación que aprovecha `evalST` y las funciones monádicas de `Control.Monad.State` para efectuar la evaluación y mostrar los resultados

```
evalwithstats :: Exp -> IO ()
```

Si la expresión a evaluar contiene una división por cero, o intenta utilizar un símbolo que no está definido en la tabla de símbolos, es suficiente que emita un mensaje de error usando `error`.

```
ghci> evalwithstats ex6
Result: *** Exception: divide by zero
ghci> evalwithstats (Var "meh")
evalwithstats (Var "meh")
Result: *** Exception: Var meh not found
```

## Evaluador con traza de operaciones (3 puntos)

Se desea implantar un evaluador de expresiones que sea capaz de registrar los resultados de las operaciones intermedias y consultas a la tabla de símbolos que se efectúan durante el cómputo.

El evaluador debe estar encapsulado en una función tal que pueda evaluarse en GHCi directamente, i.e.

```
ghci> evalwithlog ex4
Result: 252
Exp: Var "foo" -> Val: 42
Exp: Const 5 -> Val: 5
Exp: Mul (Var "foo") (Const 5) -> Val: 210
Exp: Const 2 -> Val: 2
Exp: Const 21 -> Val: 21
Exp: Mul (Const 2) (Const 21) -> Val: 42
Exp: Add (Mul (Var "foo") (Const 5)) (Mul (Const 2) (Const 21)) -> Val: 252
```

Para conseguirlo, es necesario que Ud. provea:

- Un combinador monádico aprovechando `Control.Monad.Writer` que complete la evaluación de las expresiones acarreando la información sobre cómputos intermedios

  ```
  evalWR :: {- Firma para aprovechar Control.Monad.Writer -}
  ```

  *Debe* utilizar `Data.Sequence` como estructura de acumulación para los resultados intermedios.

- La función principal de evaluación que aprovecha `evalWR` y las funciones monádicas de `Control.Monad.Writer` para efectuar la evaluación y mostrar el registro de operaciones intermedias

  ```
  evalwithlog :: Exp -> IO ()
  ```

Si la expresión a evaluar contiene una división por cero, o intenta utilizar un símbolo que no está definido en la tabla de símbolos, es suficiente que emita un mensaje de error usando `error`.

```
ghci> evalwithlog ex6
Result: *** Exception: divide by zero
ghci> evalwithlog (Var "meh")
Result: *** Exception: Var meh not found
```

## Evaluador con recuperación de errores (3 puntos)

Para completar esta parte de la Tarea es necesario que estudie el tipo de datos `Data.Either`, sabiendo que es una instancia del Monad `Error` descrito en `Control.Monad.Error`.

Se desea implantar un evaluador de expresiones que sea capaz de indicar si el resultado de la operación ha sido exitoso, o bien si ha ocurrido algún error. En este sentido, el evaluador debe estar preparado para reaccionar antes tres errores

```
data ExpError = DivisionPorCero
              | NumeroDeMalaSuerte
              | VariableNoExiste String
              deriving (Show)
```

Los errores `DivisionPorCero` y `VariableNoExiste` permiten reportar los problemas que ya hemos enfrentado en los evaluadores anteriores, mientras que el error `NumeroDeMalaSuerte` ocurrirá *siempre* que durante el cómputo aparezca el número 13. Por lo tanto, la evaluación debe reportar el error `NumeroDeMalaSuerte` si se encuentra alguna variable con el valor 13, si aparece la constante manifiesta 13 o se obtiene 13 como resultado de algún cómputo intermedio.

Si una expresión contiene múltiples errores, el evaluador sólo debe reportar el *primer* error encontrado en la travesía *inorder* implícita de evaluación.

El evaluador debe estar encapsulado en una función tal que pueda evaluarse en GHCi directamente, i.e.

```
ghci> evalwithcare ex4
Result: 252
ghci> evalwithcare (Const 13)
Rayos: NumeroDeMalaSuerte
ghci> evalwithcare (Div (Const 42) (Sub (Const 4) (Const 4))
Rayos: DivisionPorCero
ghci> evalwithcare (Var "meh")
Rayos: VariableNoExiste "meh"
```

Para conseguirlo, es necesario que Ud. provea:

- Una instancia de `ExpError` en la clase `Error`.
- Un combinador monádico aprovechando `Data.Either` que complete la evaluación de las expresiones reportando errores si los encontrara

  ```
  evalEX :: {- Firma para aprovechar Data.Either -}
  ```

- La función principal de evaluación que aprovecha `evalEX` y las funciones monádicas de `Control.Monad.Error` y `Data.Either` para efectuar la evaluación y mostrar el resultado exitoso o con el primero error encontrado

```
evalwithcare :: Exp -> IO ()
```

## Consideraciones

Note que *todos* los tipos de datos participantes en el flujo de cómputo son instancias de Monad por lo tanto, todos los combinadores deben ser absolutamente monádicos. En otras palabras, está **mal** hacer *pattern-matching* para determinar si algo es `Just` o `Nothing`, o bien `Left` o `Right`.

Las librerías `Data.Either` y `Data.Maybe` tienen combinadores monádicos – en sus respectivos Monad, obviamente – que les permiten evitar la escritura de *pattern matching*.