CIS 552: Advanced Programming

HW 3 # Monads

```
-- Advanced Programming, HW 3
-- by <YOUR NAME HERE> <pennid> (and  <YOUR PARTNERS NAME> <pennid>)
```

Submission Instructions

To complete this homework, download the nonliterate version of the file and answer each question, filling in code where noted (where it says "TBD"). Your code must typecheck against the given type signatures. Also remember to add your own tests to this file to exercise the functions you write, and make your own 'main' function to run those tests.

Also, this assignment should be done in PAIRS. Only one version of the homework should be submitted from each group.

```
{-# OPTIONS -Wall -fwarn-tabs -fno-warn-type-defaults #-}
{-# LANGUAGE NoImplicitPrelude #-}

module Main where

import Prelude hiding (mapM)
import Data.Char (isAlpha, toUpper)
import Data.Map (Map)
import qualified Data.Map as Map
import Control.Monad.State hiding (when, mapM, foldM)
import Test.HUnit hiding (State)

main :: IO ()
main = return ()
```

Functors and Monads

```
-- Problem 0
```

The following parameterized data structure, called a sequence,

```
data Seq a = Nil | Single a | Concat (Seq a) (Seq a)
```

is a list optimized for concatenation. Indeed, two sequences can be concatenated in constant time, using the `Concat` constructor.

```
--  (a)
```

It is also possible to convert a sequence into a normal list, which is useful for displaying sequence values concisely.

```
instance Show a => Show (Seq a) where
```

```
   show xs = show (toList xs)
```

However, the implementation of toList below is inefficient for left-biased sequences. For example, those of the form:

```
Concat (Concat (Concat (Concat (Single a)
                               (Single b))
                       (Single c))
               (Single d))
        (Single e))
```

In converting such sequences to lists, the append operation will need to traverse the intermediate result lists multiple times. (Recall the implementation of (++) is not constant time---it must loop over its first argument.)

Your first job is to replace this version of toList with a more efficient version.

```
toList :: Seq a -> [a]
toList Nil = []
toList (Single x)     = [x]
toList (Concat s1 s2) = toList s1 ++ toList s2

t0a :: Test
t0a = toList (Concat (Concat (Concat (Concat (Single 0)
                                             (Single 1))
                                     (Single 2))
                             (Single 3))
                     (Single 4)) ~?= [0,1,2,3,4]

-- (b)
```

Complete the Functor and Monad instances for sequences. Because we're practicing, do not use 'toList' in your implementation.

```
instance Functor Seq where
   fmap _ _  = error "TBD"

instance Monad Seq where
   return = error "TBD"

   _ >>= _ = error "TBD"
```

The functor and monad instance for sequences should be the equivalent to the ones for lists. More formally, we can say that the following list equalities should hold, no matter what values are used for f,s,x,m and k.

```
  toList (fmap f s) == fmap f (toList s)
     where s :: Seq a
           f :: a -> b

  toList (return x) == return x
     where x :: a

  toList (m >>= k) == toList m >>= (toList . k)
     where m :: Seq a
           k :: a -> Seq b
```

Write (at least) three test cases that test these three identities.

The drawback of sequences is that accessing the first element may not be a constant time operation. Nevertheless, it is possible to write a concise function to find this element, taking advantage of the fact that `Maybe` is a member of the `MonadPlus` type class. Read this wikipage for more information, and write this operation as succinctly as you can.

```
first :: Seq a -> Maybe a
first = error "TBD"
```

Maybe Monad Practice

```
-- Problem 1
```

Rewrite the map2 function from homework 1 so that it is more "picky". If the two input lists are different lengths, then the function should return Nothing instead of truncating one of the lists.

```
-- (a)
```

```
pickyMap2 :: (a -> b -> c) -> [a] -> [b] -> Maybe [c]
pickyMap2 = error "TBD"
```

```
-- (b)
```

Now use pickyMap2 to rewrite the transpose function so that if the inner lists are not all the same length, then transpose returns Nothing.

```
transpose :: [[a]] -> Maybe [[a]]
transpose = error "TBD"
```

```
-- (c)
```

Next rewrite map2 again so that the function passed in can also be partial. If any application of the function returns 'Nothing' then the entire result should be 'Nothing'

```
partialPickyMap2 :: (a -> b -> Maybe c) -> [a] -> [b] -> Maybe [c]
partialPickyMap2 = error "TBD"
```

List Comprehension Practice

```
-- Problem 2
-- (a)
```

Using a list comprehension, give an expression that calculates the sum of the first one hundred integer squares (i.e. 1^2 + 2^2 + ... + 100^2).

```
sumFirstHundred :: Int
sumFirstHundred = error "TBD"
```

```
-- (b)
```

A triple (x,y,z) is Pythagorean if x^2 + y^2 = z^2. Use a list comprehension to produce a list of pythagorean triples whose components are at most a given limit. For example, pyths 10 returns [(3,4,5), (4,3,5), (6,8,10), (8,6,10)]

```
pyths :: Int -> [(Int,Int,Int)]
pyths = error "TBD"
```

```
-- (c)
```

Read about the Sieve of Sundaram and implement it using (nested) list comprehensions. You do not need to worry about the running time and memory usage, instead make sure that your code clearly implements the algorithm described on that page.

```
sieveSundaram :: Int -> [Int]
sieveSundaram = error "TBD"
```

General Monadic Functions

```
-- Problem 3

-- (a) Define another monadic generalization of map (do not use any
--     functions defined in Control.Monad for this problem).

mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM = error "TBD"

safeUpper :: Char -> Maybe Char
safeUpper x = if isAlpha x then Just (toUpper x) else Nothing

t3a :: Test
t3a = TestList [ mapM safeUpper "sjkdhf"  ~?= Just "SJKDHF",
                 mapM safeUpper "sa2ljsd" ~?= Nothing ]

-- (b) Define a monadic generalization of foldr (again, do not use any
--     functions defined in Control.Monad for this problem).

foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
foldM = error "TBD"

t3b :: Test
t3b = TestList [ addEven [1,2,3]  ~=? Nothing,
                 addEven [2,4]    ~=? Just 6]

addEven :: [Int] -> Maybe Int
addEven xs = foldM f 0 xs where
               f x y | even x    = Just (x + y)
                     | otherwise = Nothing
```

An Interpreter for WHILE

```
-- Problem 4
```

In this problem, you will use monads to build an evaluator for a simple imperative language, called WHILE. In this language, we will represent different program variables as

```
type Variable = String
```

Programs in the language are simply values of the type

```
data Statement =
    Assign Variable Expression          -- x = e
  | If Expression Statement Statement   -- if (e) {s1} else {s2}
  | While Expression Statement          -- while (e) {s}
  | Sequence Statement Statement        -- s1; s2
  | Skip                                -- no-op
  deriving (Eq, Show)
```

where expressions are variables, constants or binary operators applied to sub-expressions

```
data Expression =
    Var Variable                        -- x
  | Val Value                           -- v
  | Op  Bop Expression Expression
  deriving (Eq, Show)
```

and binary operators are simply two-ary functions

```
data Bop =
    Plus     -- +  :: Int  -> Int  -> Int
  | Minus    -- -  :: Int  -> Int  -> Int
  | Times    -- *  :: Int  -> Int  -> Int
  | Divide   -- /  :: Int  -> Int  -> Int
  | Gt       -- >  :: Int -> Int -> Bool
  | Ge       -- >= :: Int -> Int -> Bool
  | Lt       -- <  :: Int -> Int -> Bool
  | Le       -- <= :: Int -> Int -> Bool
  deriving (Eq, Show)

data Value =
    IntVal Int
  | BoolVal Bool
  deriving (Eq, Show)
```

We will represent the *store* i.e. the machine's memory, as an associative map from `Variable` to `Value`

```
type Store = Map Variable Value
```

Note: we don't have exceptions (yet), so if a variable is not found (eg because it is not initialized) simply return the value `0`. In future assignments, we will add this as a case where exceptions are thrown (the other case being type errors.)

We will use the standard library's `State` monad to represent the world-transformer. Intuitively, `State s a` is equivalent to the world-transformer `s -> (a, s)`. See the above documentation for more details. You can ignore the bits about `StateT` for now.

Expression Evaluator

First, write a function

```
evalE :: Expression -> State Store Value
```

that takes as input an expression and returns a state-transformer that returns a value. Yes, right now, the transformer doesn't really transform the world, but we will use the monad nevertheless as later, the world may change, when we add exceptions and such.

Again, we don't have any exceptions or typechecking, so the interpretation of an ill-typed binary operation (such as '2 + True') should return always 0.

Hint: The value `get` is of type `State Store Store`. Thus, to extract the value of the "current store" in a variable `s` use `s <- get`.

```
evalE (Var _)     = error "TBD"
evalE (Val _)     = error "TBD"
evalE (Op _ _ _) = error "TBD"
```

Statement Evaluator

Next, write a function

```
evalS :: Statement -> State Store ()
```

that takes as input a statement and returns a world-transformer that returns a unit. Here, the world-transformer should in fact update the input store appropriately with the assignments executed in the course of evaluating the `Statement`.

Hint: The value `put` is of type `Store -> State Store ()`. Thus, to "update" the value of the store with the new store `s'` do `put s`.

```
evalS (While _ _)        = error "TBD"
evalS Skip               = error "TBD"
evalS (Sequence _ _   ) = error "TBD"
evalS (Assign _ _)      = error "TBD"
evalS (If _ _ _ )       = error "TBD"
```

In the `If` and `While` cases, if `e` evaluates to a non-boolean value, just skip. Finally, write a function

```
execS :: Statement -> Store -> Store
execS = error "TBD"
```

such that `execS stmt store` returns the new `Store` that results from evaluating the command `stmt` from the world `store`. **Hint:** You may want to use the library function

```
execState :: State s a -> s -> s
```

When you are done with the above, the following function will "run" a statement starting with the `empty` store (where no variable is initialized). Running the program should print the value of all variables at the end of execution.

```
run :: Statement -> IO ()
run stmt = do putStrLn "Output Store:"
              putStrLn $ show $ execS stmt Map.empty
```

Here are a few "tests" that you can use to check your implementation. (You do not need to fit these test cases into 80 columns.)

```
w_test :: Statement
w_test = (Sequence (Assign "X" (Op Plus (Op Minus (Op Plus (Val (IntVal 1)) (Val (IntVal 2))) (Val (

w_fact :: Statement
w_fact = (Sequence (Assign "N" (Val (IntVal 2))) (Sequence (Assign "F" (Val (IntVal 1))) (While (Op (
```

When you are done, the following tests should pass:

```
t4a :: Test
t4a = execS w_test Map.empty ~?=
        Map.fromList [("X",IntVal 0),("Y",IntVal 10)]

t4b :: Test
t4b = execS w_fact Map.empty ~?=
        Map.fromList [("F",IntVal 2),("N",IntVal 0),("X",IntVal 1),("Z",IntVal 2)]
```

```
ghci> run w_test
Output Store:
fromList [("X",IntVal 0),("Y",IntVal 10)]

ghci> run w_fact
Output Store:
fromList [("F",IntVal 2),("N",IntVal 0),("X",IntVal 1),("Z",IntVal 2)]
```

Credit: Original version of Problem 4 from UCSD

News :

Welcome to CIS 552!
See the home page for basic information about the course, the schedule for the lecture notes and assignments, the resources for links to the required software and online references, and the syllabus for detailed information about the course policies.

Links :

- Piazza
- Haskell.org
- GHC manual
- [illegible]