The University of Hong Kong

COMP3258: Functional Programming

Assignment 3 (IO and Monads)

Deadline: 23:55, November 29th, 2020 (HKT)

---

1 Tic Tac Toe

1.1 Introduction

Problem 1. (25 pt.) A tic-tac-toe game is two players (`X` and `O`) playing on a 3 by 3 grid in turn, whichever player first assembles 3 pieces in a line (row, column or diagonal) wins. And you are going to implement this game in haskell with the following specification:

```
tictactoe :: IO ()
tictactoe = undefined
```

`tictactoe` is an interactive IO operation, when invoked, it starts the game with the following flow:

Step 1. Printing the board The board MUST be printed (on the screen / command line) in the following format:

An empty board looks like this:

```
.---.---.---.
|   |   |   |
.---.---.---.
|   |   |   |
.---.---.---.
|   |   |   |
.---.---.---.
```

whereas a non-empty board looks like this, by replacing the central character of the cell with `X` (capital) or `O` (capital letter O, NOT number zero).

```
.---.---.---.
| X | O |   |
.---.---.---.
|   | O |   |
.---.---.---.
|   |   |   |
.---.---.---.
```

PLEASE PAY ATTENTION TO THE FORMAT CHARACTER BY CHARACTER.

Step 2. Move Then, your program should print `X MOVE` or `O MOVE` at a new line, according to current moving player.

Then it reads *one line* of user input, which consists of a pair of coordinate <X> <Y>. where <X> denotes the number of row, <Y> denotes the number of column, both indices are base 1 (`1 2` denotes the first row and the second column, NOT the second row and the third column)

A correct user input consists of two valid coordinates separated by an arbitrary amount of whitespaces.

For example the following lines are valid inputs:

```
1 2
2   3
```

However the following are not (`#` are followed by explanations):

```
2      # only one coordinate
12 2   # out of range coordinate (12)
1 2 xy # xy shouldn't appear
abdsaf # input that doesn't make any sense
```

Besides, the attempt of making a move on a non-empty cell is considered invalid input.

Upon invalid input, your program should print `INVALID POSITION` followed by a newline, reads user input again. This process repeats until a correct input is read.

Step 3. Loop If the game result is determined, then a message is printed and the program returns.

- If player `X` wins, prints `X WINS`.
- If player `O` wins, prints `O WINS`.
- If the board is full, but neither player wins, prints `DRAW`.

Otherwise return to step 1, printing the current state of the board.

1.2 Further Clarification

- Two sample inputs and outputs are attached in the appendix of this file.
- Player `O` always moves first.
- Your program should handle incorrect user inputs as specified in "Step 2".
- The grading of this question would be mostly based on the exact match of the command line output of your program and standard output, so please pay extra attention to the output format. Don't let small typos cost you the entire question.
- So for your convenience (and mine), all the printed letters are uppercase.
- Try your best to keep your code clean, the clarity of code and the familiarity of IO operation and monad shown in the code would probably be considered.

2 Monads

2.1 The `Transform` Monad

Problem 2. (25 pt) Implement the `Monad` instance of datatype `Transform`, and an important function `next`

The definition of `Transform` is given:

```
newtype Transform a b = Transform { getTransform :: (b, a -> a) }

instance Monad (Transform a) where
```

```haskell
  -- return :: b -> Transform a b
  return b = undefined

  -- (>>=) :: Transform a b -> (b -> Transform a c) -> Transform a c
  (>>=) = undefined

next :: (a -> a) -> Transform a ()
next = undefined
```

The `Functor` instance and the `Applicative` instance have been given in the template. Changing the definition of them is allowed if you know what you are doing, but generally discouraged.

2.1.1 General Behavior

`Transform` is a data type that's equivalent to a pair of type `b`, and a transform function of type `a`. The `Functor`, `Applicative` and `Monad` are all defined with respect the type variable `b`, and `a` is relatively fixed in the definition (similar to `Either a b`).

Inside a `do` notation or monadic computation, the `Transform` monad "accumulates" the transformation function of `a` with the function `next` (and of course, with the help of the behavior of `>>=`), and focus on the computation of `b` (In a `do` block, assuming the type on the right-hand-side of the left arrow `<-` is `Transform a b`, the type of pattern of the left-hand-side would be `b`).

We have the following example:

```haskell
countedFibonacci :: Int -> Transform Int Int
countedFibonacci 0 = return 0
countedFibonacci 1 = return 1
countedFibonacci n = do
   a <- countedFibonacci (n - 1)
   next (+1)
   b <- countedFibonacci (n - 2)
   return $ a + b
```

The example above computes the *nth* term in Fibonacci sequence, while accumulating the number of times the recursive case of the function has been called.

Here the computation is about the nth term, so the type of computation result is `Int` (the second type parameter). The transformation we want to accumulate is adding the number of times for which the recursion is called, which is `Int -> Int` (so the first type parameter is `Int`).

The call of `next (+1)` adds `(+1)` (which is a function of `Int -> Int`) on top of the existing transformation function, but itself produce no useful value for the computation (it doesn't contribute to the computation of nth term of Fibonacci at all).

We provide the following two utility functions in the template that extract the different parts of `Transform`:

```
-- gets the computation result
evalTransform :: Transform a b -> b
evalTransform = fst . getTransform

-- gets the transformation function
runTransform :: Transform a b -> a -> a
runTransform = snd . getTransform
```

And the call of `countedFibonacci` goes as:

```
> evalTransform (countedFibonacci 10)
55
> runTransform (countedFibonacci 10) 0
              initial value apply to ^
              the transformation
88
> evalTransform (countedFibonacci 16)
987
> runTransform (countedFibonacci 16) 0
1596
```

2.1.2 `Tree` and `Transform`

The following definition of the `Tree` datatype is given:

```
data Tree a = Leaf | Branch (Tree a) a (Tree a)
```

Problem 3. (20 pt) Implement the following two functions with `Transform`

```haskell
tFoldl :: (b -> a -> b) -> b -> Tree a -> Transform [a] b
tFoldl = undefined

tToListWith :: (b -> a -> b) -> Tree a -> Transform b [a]
tToListWith = undefined
```

`tFoldl` is a left-fold of the tree data type similar to `foldl` of lists. But meanwhile, it records a transformation function (`[a] -> [a]`) that produces an in-order traversal result of the tree when applying to `[]` (empty list).

`tToListWith` behaves almost the same as `tFoldl` , except that it switches the focus of the computation and transformation function. The main computation result `[a]` is an in-order traversal of the tree. The transformation function `b -> b`, when applying to an initial value `z` , the transformation function produces the left-fold result of the tree, with the folding function and initial value.

```haskell
-- shape of t:
--        4
--      /   \
--     2     6
--    / \   /
--   1   3 5
> t = Branch (Branch (Branch Leaf 1 Leaf) 2 (Branch Leaf 3 Leaf))
             4 (Branch (Branch Leaf 5 Leaf) 6 Leaf)

> evalTransform (tFoldl (\b x -> b ++ show x) "" t)
"123456"
> runTransform (tToListWith (\b x -> b ++ show x) t) ""
"123456"
-- we don't use the fold functions in the following two samples
> evalTransform (tToListWith undefined t)
[1,2,3,4,5,6]
> runTransform (tFoldl undefined undefined t) []
[1,2,3,4,5,6]
```

2.1.3 Hints and Clarifications

- Your monad definition should satisfy the three monad laws mentioned in the next section, but you don't need to worry about that too much.

Your focus should be making the monad behave as intended.

- With correct implementation, the definitions of both monad methods are VERY simple. So focus on understanding what's going on in the examples and samples, the implementation should be fairly easy afterward.
- The samples are generated from my implementation, which may not be correct. If your results are different from mine, and you *strongly* believe that your implementation is correct, please let me (Alvin) know.

2.2 Composing List and `Maybe`

Problem 4. (15 + 15 pt) Implement the Monad Instance of the 2 forms of Composition of `[]` and `Maybe`.

```
newtype LM a = LM { getLM :: [Maybe a] }

instance Monad LM where
  -- return :: a -> LM a
  return a = undefined

  -- (>>=) :: LM a -> (a -> LM b) -> LM b
  (>>=) = undefined

newtype ML a = ML { getML :: Maybe [a] }

instance Monad ML where
  -- return :: a -> ML a
  return a = undefined

  -- (>>=) :: ML a -> (a -> ML b) -> ML b
  (>>=) = undefined
```

The `Functor` instances and the `Applicative` instances have been given in the template. Changing the definition of them is allowed if you know what you are doing, but generally discouraged.

There would be no sample input and output for this question. Any definition that satisfies the three monad laws is accepted:

```
x :: a
m :: M a
```

```haskell
f :: a -> M b
g :: b -> M c

-- Left Identity of return
return x >>= f = f x

-- Right Identity of return
m >>= return = m

-- Associativity of (>>=)
m >>= (\x -> f x >>= g) = (m >>= f) >>= g
```

Hints and Clarifications

- You should probably be very familiar with the monadic behavior of `[]` (List) and `Maybe` before solving this problem.
- Again, with a correct implementation, all the methods should be fairly simple, so the focus of this question is on the thought process instead of the implementation.
- Thinking about why the following stupid definition is wrong might be a good start:

```haskell
instance Monad LM where
  return a = LM []
  m >>= f = LM []

instance Monad ML where
  return a = ML Nothing
  m >>= f = ML Nothing
```

- Strictly speaking, failing to satisfy one of the three rules will get you 0 score for the corresponding monad instance, because both stupid implementations above only violate one rule.
- Try your best to simplify your definition with existing combinators in library, the conciseness of the definition would probably be considered.

3 Submission Caveats

3.1 Grading

- The grading of every question is done by comparing the outputs against certain test cases automatically by the grading script, where each test case contributes to some points unless otherwise specified.
- We are likely to have a manual process looking at the taste of the code, the share of the score hasn't been determined. But if your program fails most of the test cases, it's likely that you also don't get this portion of the score.
- Time complexity of implementation is NOT assumed. However, your programs should terminate with relatively small input in a "reasonable" amount of time.

3.2 Implementation

- You are encouraged to define your own auxiliary functions or data types to solve the problems, as long as you implement the functions and data types explicitly required by the questions.
- You are NOT allowed to add additional source files besides the three in the templates
- You are NOT allowed to add any additional `import` statement, the imports in the templates should be more than enough for you. However you are allowed to remove `import` statements in the template in case of version compatibility issues, but it's strongly discouraged so don't do that unless it's absolutely necessary.
- With the said restrictions, you are free to use any library functions and language extensions supported by ghc 8.10 or lower.
- Ideally, your ghc compiler should be version 8.6.1 or above.

3.3 Submission

- Rename the directory containing template files to `A3_XXX` where `XXX` is replaced by your student number. Pack the directory into a *zip* file named `A3_XXX.zip` when submitting (other compression formats `rar`, `7z` etc are NOT accepted), i.e. the zip file `A3_XXX.zip` contains a directory `A3_XXX`, which contains *exactly* three template source files with their names unchanged.

- Please submit your assignment to moodle before deadline.

4 Appendix: Sample Input / Output of Question 1

We show some sample input and output, inputs are entered line by line whenever the program instruct people to enter a line of input. Please pay attention to the outputs and positions of newlines etc.

1 Inputs:

```
2 2
1 1
1 2
2 3
3 2
```

Command line outputs:

```
.---.---.---.
|   |   |   |
.---.---.---.
|   |   |   |
.---.---.---.
|   |   |   |
.---.---.---.
O MOVE
.---.---.---.
|   |   |   |
.---.---.---.
|   | O |   |
.---.---.---.
|   |   |   |
.---.---.---.
X MOVE
.---.---.---.
| X |   |   |
```

```
.---.---.---.
|   | O |   |
.---.---.---.
|   |   |   |
.---.---.---.
O MOVE
.---.---.---.
| X | O |   |
.---.---.---.
|   | O |   |
.---.---.---.
|   |   |   |
.---.---.---.
X MOVE
.---.---.---.
| X | O |   |
.---.---.---.
|   | O | X |
.---.---.---.
|   |   |   |
.---.---.---.
O MOVE
.---.---.---.
| X | O |   |
.---.---.---.
|   | O | X |
.---.---.---.
|   | O |   |
.---.---.---.
O WINS
```

2 Inputs:

```
2 2
1 1
1 1
1 1 2 2
1 2 3 1
3 2
1 2
1 3
3 1
```

```
2 1
2 3
3 3
```

Commandline output:

```
.---.---.---.
|   |   |   |
.---.---.---.
|   |   |   |
.---.---.---.
|   |   |   |
.---.---.---.
O MOVE
.---.---.---.
|   |   |   |
.---.---.---.
|   | O |   |
.---.---.---.
|   |   |   |
.---.---.---.
X MOVE
.---.---.---.
| X |   |   |
.---.---.---.
|   | O |   |
.---.---.---.
|   |   |   |
.---.---.---.
O MOVE
INVALID POSITION
INVALID POSITION
INVALID POSITION
.---.---.---.
| X |   |   |
.---.---.---.
|   | O |   |
.---.---.---.
|   | O |   |
.---.---.---.
X MOVE
.---.---.---.
| X | X |   |
```

```
.---.---.---.
|   | O |   |
.---.---.---.
|   | O |   |
.---.---.---.
O MOVE
.---.---.---.
| X | X | O |
.---.---.---.
|   | O |   |
.---.---.---.
|   | O |   |
.---.---.---.
X MOVE
.---.---.---.
| X | X | O |
.---.---.---.
|   | O |   |
.---.---.---.
| X | O |   |
.---.---.---.
O MOVE
.---.---.---.
| X | X | O |
.---.---.---.
| O | O |   |
.---.---.---.
| X | O |   |
.---.---.---.
X MOVE
.---.---.---.
| X | X | O |
.---.---.---.
| O | O | X |
.---.---.---.
| X | O |   |
.---.---.---.
O MOVE
.---.---.---.
| X | X | O |
.---.---.---.
| O | O | X |
.---.---.---.
| X | O | O |
```

```
.---.---.---.
DRAW
```