

CSCC24 2020 Winter – Assignment 2  
Due: Wednesday, February 26, midnight  
This assignment is worth 10% of the course grade.

In this assignment, a domain-specific monadic type class is given, i.e., there are domain-specific methods, along with the usual Monad, Applicative, and Functor methods as connectives. You will work on both sides of the fence:

- You will implement an instance of this monad class. (This means a model or representation or semantics.)
- You will implement some programs in the domain. They use only the methods, and so they are polymorphic in the type class, decoupled from particular instances. The same program can then be instantiated to different instances for different purposes.

As usual, you should also aim for reasonably efficient algorithms and reasonably organized, comprehensible code.

## Decision Trees

In this assignment, a decision tree is a binary tree in which an internal node stores a probability and two child subtrees, and a leaf stores an outcome.

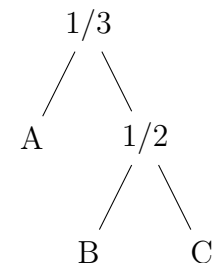
```
data DecisionTree a = Tip a
                    | Choose Rational (DecisionTree a) (DecisionTree a)
```

Imagine walking down one such tree. When at an internal node storing probability  $p$ , you randomly choose whether to go left or right, with probability  $p$  of choosing left, and probability  $1 - p$  of choosing right. When you hit a leaf node, it specifies your outcome.

(Outside this assignment, decision trees are more general: Internal nodes may have conditions instead of probabilities, and may have more child subtrees.)

Here is an example; it represents equal probability ( $1/3$ ) of obtaining the 3 outcomes. (There are other trees representing the same distribution.)

```
abcTree =
  Choose (1/3)
    (Tip 'A')
    (Choose (1/2) (Tip 'B') (Tip 'C'))
```



### Question 1: Expected values and probabilities (2 marks)

A random variable is, formally, a function from outcomes to real numbers. (There is an informal explanation that doesn't help in this assignment.) For example, imagine a gambling game (as usual) in which the 3 outcomes are A, B, and C, and you win \$3 for A, lose \$1 for B, and lose \$4 for C. This rule is represented by the following random variable (again, a function):

```

abcProfit 'A' = 3
abcProfit 'B' = -1
abcProfit 'C' = -4

```

Given a random variable  $rv$  and a decision tree  $t$ , the expected value (of  $rv$  under  $t$ ) is the sum, over all possible outcomes  $a$ , of (probability of getting  $a$ )  $\times$   $rv(a)$ . As an application, the expected value of  $abcProfit$  under  $abcTree$  is your average gain/loss per game if you play this game many times in the long run, which should come out as  $-2/3$ .

Implement an algorithm for computing expected values. To maximize convenience for users (therefore annoyance for you!), it is highly polymorphic in the number type, not always the rational numbers.

```

expectedValue :: Fractional q => (t -> q) -> DecisionTree t -> q

```

As another application or corollary, from the ability of computing expected values of arbitrary random variables, we get the ability of computing probabilities of arbitrary events. (An event is a predicate on the outcomes.) The trick is to conjure a Bernoulli random variable, i.e., a function that maps desired outcomes to 1 and other outcomes to 0. See the provided function

```

probability :: Fractional q => (t -> Bool) -> DecisionTree t -> q

```

For example, the probability of not getting B can be computed by

```

probability (\x -> x/= 'B') abcTree

```

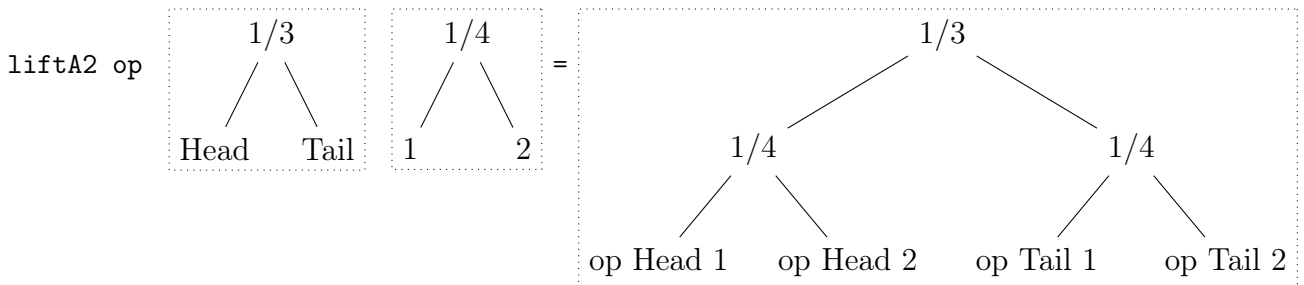
which should come out as  $2/3$ .

## Question 2: Decision tree type as a random monad type (10 marks)

Make the decision tree type an instance of Functor, Applicative, and Monad. Implement the relevant methods.

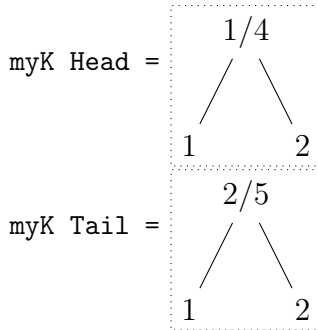
Informally, `fmap` simply converts the outcomes while preserving probabilities. `liftA2` is like performing two random experiments (represented by two trees) and combining outcomes by a binary operator. Bind (`>>=`) is like performing the first experiment and using its outcome to determine what second experiment to perform or what second distribution to use. `pure` and `return` give a single outcome with probability 1, i.e., leaf node.

Here are some examples in terms of actual trees. For `liftA2`:

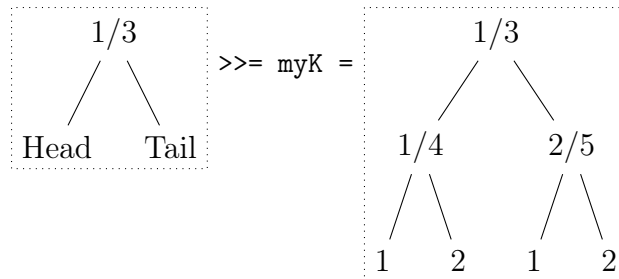


It is true that the two component experiments are treated as independent, i.e., the joint probability is simply multiplying the two respective probabilities. Also recall the `rr` function in a recent lab exercise; when used in this context, it repeats a given experiment  $n$  times.

For `bind (>>=)`: If `myK` is defined as:



then



It is true that bind is how you can make a later experiment depend on an earlier one.

Moreover, the decision tree type is also an instance of the following special-purpose type class:

```

class Monad m => MonadRandom m where
  choose :: Rational -> m a -> m a -> m a

```

Informally, an instance type  $T$  is a type representing random experiments, and a value  $e :: TA$  represents a random experiment with sample space  $A$  (the outcomes have type  $A$ ).

*choose*  $p e_1 e_2$  is the primitive for randomization: randomly chooses to pursue sub-experiment  $e_1$  or  $e_2$ , with probability  $p$  of pursuing  $e_1$ , and probability  $1 - p$  of pursuing  $e_2$ . The methods of Functor, Applicative, and Monad are then for producing individual outcomes and building up more complex experiments.

This narrative works for the decision tree type; it also works for other representation types. For example, the bonus question shows one that focuses on expected values. There is another that focuses on using a pseudo-random number generator for Monte Carlo runs; this one is outside the scope of this assignment.

Implement *choose* for the decision tree type (in the most obvious way).

### Question 3: Decoupling from representations (6 marks)

Randomized algorithms can be expressed solely using the methods of MonadRandom, Monad, etc., abstracting away from decision trees or any other representation. For example, the two examples in the previous question can be abstractly expressed as

```

ex1 :: MonadRandom m => m (Coin, Integer)
ex1 = liftA2 (\x y -> (x,y))
          (choose (1/3) (pure Head) (pure Tail))
          (choose (1/4) (pure 1) (pure 2))

```

```

ex2 :: MonadRandom m => m Integer
ex2 = choose (1/3) (return Head) (return Tail)
    >>= \x -> case x of
        Head -> (choose (1/4) (return 1) (return 2))
        Tail -> (choose (2/5) (return 1) (return 2))

```

They are polymorphic in the experiment representation type. They can be instantiated to the decision tree type to generate the example trees in the previous question, suitable for various analyses; but they can also be instantiated to other representations optimized for other purposes, e.g., expected values or Monte Carlo runs. When this technique is applied to business applications, the following analogies can be drawn:

randomized algorithm	business code
instantiate for analyses	testing or validation
instantiate for Monte Carlo runs	production runs

It is now your turn to express some randomized algorithms!

1. Given a non-empty list, randomly pick an item in the list, with equal probability over all items in the list.

```
uniform :: MonadRandom m => [a] -> m a
```

What if the list contains duplicates, e.g., [7,3,7]? Then the 1st occurrence of 7 has probability 1/3, and the 2nd occurrence also has probability 1/3.

If instantiated to decision trees, the tree height should be at most  $\lceil \lg(\text{list length}) \rceil + 1$ . Any correct tree within the height bound is accepted.

2. Randomized Hangman: Play the Hangman game by random guessing. Initially, a word is given and is to be guessed. You are allowed  $k$  turns. In each turn, pick a letter that has not been picked yet, and see whether it occurs in the word one or more times. You lose if you haven't hit all letters in the word after  $k$  turns; you win otherwise.

```

data WinLose = Lose | Win
hangman :: MonadRandom m => String -> Int -> String -> m WinLose

```

`hangman word k letters` plays the game guessing `word`, in `k` turns, and picks from `letters`. Assume that every letter in `word` is in `letters`, `letters` does not contain duplicates, and there are only lowercase letters.

## Bonus Question (extra 20%)

Expected value of a random variable  $X$  is defined one way or another in this form:

$$E(X) = \dots X \dots$$

This invites us to think of  $E$  itself as a function that maps random variables to numbers, i.e.,  $E : (A \rightarrow \mathbb{R}) \rightarrow \mathbb{R}$ . Let's give a name to this type (and use rationals for simplicity):

```
data Ev a = MkEv ((a -> Rational) -> Rational)
```

It turns out that `Ev` is an instance of `MonadRandom`. Implement the methods. You must also include a proof for one of `liftA2`, `<*>`, or `>>=`. In the proof, you may assume that all sample spaces are finite sets. You receive no bonus marks if you don't include the proof.

End of questions.