Programming in Haskell – Homework Assignment 6

UNIZG FER, 2016/2017

Handed out: January 06, 2017. Due: January 17, 2017 at 17:00

Note: Define each function with the exact name and the type specified. You can (and in most cases you should) define each function using a number of simpler functions. Provide a type signature above each function definition and comment the function above the type signature. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values. Use the `error` function for cases in which a function should terminate with an error message. Problems marked with a star (⋆) are optional.

Each problem is worth a certain number of points. The points are given at the beginning of each problem or subproblem (if they are scored independently). These points are scaled, together with a score for the in-class exercises, if any, to 10. Problems marked with a star (⋆) are scored on top of the mandatory problems, before scaling. The score is capped at 10, but this allows for a perfect score even with problems remaining unsolved.

1. *(3 pts)* Create a simple utility for working with CSV (comma-separated value) files. These are simple textual files where fields are delimited with a character (usually a comma or a semicolon). We will use the types provided below to define a CSV document and implement basic operations over them. We require that the CSV document is well-formed, i.e., that it contains an equal number of fields per row. You may assume separators will always be of length 1, even though they are strings.

   Note: You can of course define your own custom datatype to help you solve this task, but please implement the following functions exactly as they are defined here. This greatly helps us with the grading process. Implementing these functions should be easy in terms of making your datatype instance of the `Show` and `Read` typeclasses and then writing a wrapper function around them. Doing this is completely optional.

   ```
   type Separator = String
   type Document  = String
   type CSV       = [Entry]
   type Entry     = [Field]
   type Field     = String

   doc = "John;Doe;15\nTom;Sawyer;12\nAnnie;Blake;20"

   brokenDoc = "One;Two\nThree;Four;Five"
   ```

   (a) Define a function `parseCSV` that takes a separator and a string representing a CSV document and returns a CSV representation of the document.

   ```
   parseCSV :: Separator -> Document -> CSV
   parseCSV ";" doc
   ```

⇒ `[["John", "Doe", "15"], ["Tom", "Sawyer", "12"], ["Annie","Blake", "20"]]` `parseCSV "," doc`
⇒ `error "The separator ‘,’ does not occur in the text"` `parseCSV ";" brokenDoc` ⇒ `error "The CSV file is not well-formed"`

(b) Define a function `showCSV` that takes a separator and a CSV representation of a document and creates a CSV string from it.

```
showCSV :: Separator -> CSV -> Document
```

```
csv = parseCSV ";" doc
```

`showCSV ";" csv` ⇒ `"John;Doe;15\nTom;Sawyer;12\nAnnie;Blake;20"` `showCSV ";" [["One"], ["Two", "Three"]]`
⇒ `error "The CSV file is not well-formed"`

(c) Define a function `colFields` that takes a CSV document and a field number and returns a list of fields in that column.

```
colFields :: Int -> CSV -> [Field]
```

`colFields 1 csv` ⇒ `["Doe", "Sawyer", "Blake"]` `colFields 3 csv` ⇒ `error "There is no column 3 in the CSV document"`

(d) Parsing CSV without the ability to access CSV files is not very useful. Define an IO function (an action) `readCSV` that takes a file path and a separator and returns the CSV representation of the file (wrapped due to impurity). In the examples below we assume the two strings, `doc` and `brokenDoc`, were written into files `doc.csv` and `brokenDoc.csv`, respectively.

Note: You can wrap a value inside IO using either `return` or `pure` functions. The latter needs to be imported from a module in older compiler versions, so `return` is a safer bet. The `return` is a *function*, don't confuse it with the *keyword* in other languages. It just happens to be called the same, but it only serves for wrapping a value.

```
readCSV :: Separator -> FilePath -> IO CSV
```

```
readCSV ";" "doc.csv"
⇒ [["John", "Doe", "15"],
["Tom", "Sawyer", "12"],
["Annie","Blake", "20"]]
readCSV "," "doc.csv"
⇒ error "The character ‘,’ does not occur in the text"
readCSV ";" "brokenDoc.csv"
⇒ error "The CSV file is not well-formed"
```

(e) Define a function `writeCSV` that takes a separator, a file path, and a CSV document and writes the document into a file.

```
writeCSV :: Separator -> FilePath -> CSV -> IO ()
```

```
writeCSV ";" "test.csv" [["1"], ["2", "3"]]
⇒ error "The CSV file is not well-formed"
writeCSV ";" "doc.csv" (parseCSV document ";")

$ cat doc.csv
John;Doe;15
Tom;Sawyer;12
Annie;Blake;20
```

(f) Place the implementation in a separate module `CSVUtils`, in a file called `CSVUtils.hs`. Expose only the five functions and five types from previous subtasks, while any helper functions should remain hidden. Submit the file with the homework assignment.

2. *(3 pts)*

   You will implement a simple Naïve Bayes (NB) classifier that learns from data recorded in a CSV file. The file has multiple rows, the first of which is always a header row containing column names and can be ignored. The first three columns contain values for the feature vector $\vec{x}$, where the first row below the header is the first feature vector and so on. The last column of every row (save the header) is the `String` value of the decision. You only have to work with files that contain two possible decisions in the last column, thereby implementing a binary classifier. You may assume that the CSV file will always be well-formed.

   In effect, you have to implement the following formula, using the data in the CSV file as a training set:

$$y = argmax_{k \in \{0,1\}} P(C_k) \prod_{i=1}^{n} P(x_i|C_k) \tag{1}$$

   $P(C_k)$ is the *a priori* probability of class C_k occurring. It can be estimated as the percentage of decisions C_j, $j \in \{0, 1\}$ for which $j = k$. In other words, it is the percentage of decision k in the overall numbers of decisions.

   The probability $P(x_i|C_k)$ is the probability of feature x_i (that particular value of the feature at index i in the feature vector) appearing when the decision C_k is made. You can estimate it by computing the number of decisions C_k where x_i appears, divided by the overall number of decisions C_k.

   Both of these types of probabilities are learned *solely over the training set*, loaded from the CSV file.

   Let us look at an example, from file `data.csv`:

```
Weather;Busy;Week Period;Study Haskell?
Sunny;Yes;Weekend;No
Raining;No;Weekend;Yes
Sunny;No;Workday;Yes
Cloudy;Yes;Workday;No
Sunny;No;Weekend;No
Raining;No;Workday;Yes
Cloudy;Yes;Weekend;Yes
```

   We will try to classify $\vec{x}$ = `["Raining", "Yes", "Workday"]`. We will say decision C_0 is `"No"` and decision C_1 is `"Yes"`. The relevant estimated probabilities are:

$$\begin{aligned}
P(C_0) &= \frac{3}{7}\\
P(C_1) &= \frac{4}{7}\\
P(Raining|C_0) &= \frac{0}{3}\\
P(Raining|C_1) &= \frac{2}{4}\\
P(Yes|C_0) &= \frac{2}{3}\\
P(Yes|C_1) &= \frac{1}{4}\\
P(Workday|C_0) &= \frac{1}{3}\\
P(Workday|C_1) &= \frac{2}{4}
\end{aligned}$$

By combining the right-hand side of equation (1) and the probabilities we just computed, we can compute the probabilities for each class:

$$\begin{aligned}
P(C_0|\vec{x}) &= \frac{3}{7}\cdot\frac{0}{3}\cdot\frac{2}{3}\cdot\frac{1}{3} = 0\\
P(C_1|\vec{x}) &= \frac{4}{7}\cdot\frac{2}{4}\cdot\frac{1}{4}\cdot\frac{2}{4} = 0.0357
\end{aligned}$$

Since the decision for which the probability is greatest is C_1, our final decision is `"Yes"`. In case of equal probabilities, you may pick either decision.

Your task is to implement the `nbDecide` and `nbDecideAll` functions that automate the task you just saw. Use the CSV library you developed in the first task (or, if you failed to solve the task, use any CSV-parsing library for Haskell, which you can install using `cabal install`).

```
type Decision = String
caseA = ["Raining", "Yes", "Workday"]
caseB = ["Sunny", "No", "Workday"]

nbDecide :: CSV -> [String] -> Decision

-- Assuming data contains the parsed contents of data.csv
nbDecide data caseA ⇒ "Yes" nbDecide data caseB ⇒ "Yes"

nbDecideAll :: CSV -> [[String]] -> [Decision]
nbDecideAll data [caseA, caseB] ⇒ ["Yes", "Yes"]
```

3. *(2 pts)* Define a typeclass `Truthy` that defines types that can be interpreted as boolean values. The typeclass must offer two functions:

   ```
   truey :: a -> Bool
   falsey :: a -> Bool
   ```

   Provide default implementations so that the minimal definition for the `Truthy` typeclass requires the definition of only one of those functions (either). Make `Bool`, `Int` and `[a]` instances of `Truthy`. For `Int`, only 0 should be considered falsey, while for lists only an empty list should be considered falsey.

   (a) Define a function `if'` that works on instances of `Truthy` and behaves like the *if-then-else* construct.

   ```
   if' :: Truthy p => p -> a -> a -> a
   if' [1,2,3] "True" "False" ⇒ True
   if' (1 > 2) "GT" "LE" ⇒ "LE"
   if' (0 :: Int) 1 2 ⇒ 2
   ```

   (b) Define a function `assert` that takes a `Truthy` value and another argument. If the first argument evaluates to truey, it returns the second argument. Otherwise it raises an error.

   ```
   assert :: Truthy p => p -> a -> a
   assert [1,2,3] [4,5,6] ⇒ [4,5,6]
   assert [] (-2) ⇒ error "Assertion failed"
   ```

   (c) Define the `(&&&)` and `(|||)` functions that behave like the `(&&)` and `(||)` functions, but operate on `Truthy` instances instead of `Bool`.

   ```
   (&&&) :: (Truthy a, Truthy b) => a -> b -> Bool
   (|||) :: (Truthy a, Truthy b) => a -> b -> Bool

   [1] &&& [2] ⇒ True
   (1 :: Int) ||| (0 :: Int) ⇒ True
   [1,2] &&& (0 :: Int) ⇒ False
   [1,2] ||| (0 :: Int) ⇒ True
   ```

4. *(2 + 1⋆ pt(s))* We know that the `(++)` concatenation operator can be expensive, especially when repeated multiple times (e.g., in pretty-printing). For that reason, you shall implement a difference list. A difference list stores the concatenation as a computation. For that reason, all appending operations are `O(1)` as long as we are working with difference lists – the only time we truly need to perform the concatenation is when we are transforming the difference list into a list.

   (a) Implement a datatype `DiffList` that holds a concatenation computation - given a list it will hold a `function` - the concatenation of that list and another list it expects as an argument. Use record syntax to define a function `undiff` that takes a list and returns the concatenation of the `DiffList` and that list (as a list).

   ```
   data DiffList a = { undiff :: ... }
   -- assuming dl is a DiffList containing the computation [1,2]++[3]++x:
   undiff dl [4] ⇒ [1,2,3,4]
   ```

(b) Implement a function `empty` that constructs an empty `DiffList`.

```
empty :: DiffList a
undiff empty [] ⇒ []
undiff empty [2,3] ⇒ [2,3]
```

(c) Implement a function `fromList` that takes a list and returns a `DiffList` as its concatenation computation.

```
fromList :: [a] -> DiffList a
undiff (fromList [1,2,3]) [4,5] ⇒ [1,2,3,4,5]
```

(d) Implement a function `toList` that takes a `DiffList` and returns its computation, a concatenated list.

```
toList :: DiffList a -> [a]
toList $ fromList xs == xs
```

(e) Implement a function `append` that takes two `DiffLists` and combines them into a new `DiffList`. (Hint: Note that this amounts to function composition since contents of `DiffLists` are functions)

```
append :: DiffList a -> DiffList a -> DiffList a
toList $ append dl dl ⇒ [1,2,3,1,2,3]
```

(f) *(1⋆ pt)* Declare `DiffList` an instance of Monoid. Monoids have to have an associative binary operation and an identity element – that means that for the operation `op` and an identity element `null` it will hold that `x op null == null op x == x` for any `x`. A monoid also has to satisfy some laws (known as the `monoid laws`) that are derived from this. The minimal complete definition for a `Monoid` consists of two functions – `mempty`, that defines the identity element, and `mappend`, that defines the associative binary operator. Tip: don't think of a Monoid as some Haskell specific thing, it is actually a direct implementation of the monoid algebraic structure used in group theory and some other applications.

5. *(⋆)(4 pts)* You have seen how a Monad instance can be used to handle things that can fail. Another use case for a Monad is dealing with non-determinism. The simplest way to model non-deterministic computation is using a list. Think of the list as holding all possible results of some computation. When you use the well known `map` function, the function you are applying produces the outputs for all the possible inputs.

But before we get to a Monad we have to start from the top. A Monad is just a type class, for example, like Ord. When you want to implement the instance of a type class, you need to implement its functions. You may recall that for Ord, there is another requirement, you have to implement the instance for Eq. Monad also has a similiar requirement, you have to implement instances of 2 type classes: Functor and Applicative. Don't let the names scare you, we will go through each one, step by step.

Define your list type like so:

```
data MyList a = Cons a (MyList a) | Nil deriving Show
```

(a) Make your list an instance of Functor. The most common Functor instances are containers. A list is a natural fit. You could also make your Tree and

Queue types from previous homeworks an instances of Functor. The type class is defined this way:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

The idea is to lift a regular function to work inside a context, here being a list of elements.

```
instance Functor MyList where
    fmap = ...
```

```
fmap id $ Cons ’a’ (Cons ’b’ (Cons ’c’ Nil))
⇒ Cons ’a’ (Cons ’b’ (Cons ’c’ Nil)) fmap (+1) $ Cons 1 (Cons 2 (Cons
3 Nil))
⇒ Cons 2 (Cons 3 (Cons 4 Nil))
```

For easier use, `fmap` has an infix operator defined: `<$>`. You can think of it as regular application, `$`, but inside some context.

```
(’A’:) <$> Cons "gda" (Cons "rc" Nil)
⇒ Cons "Agda" (Cons "Arc" Nil)
```

(b) Functor instance is useful when you want to use a regular function on your list elements. But what if you have a list of functions and want to apply the functions to elements of another list? This is where the Applicative type class comes in.

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

As you can see from the definition, it requires the type to be an instance of Functor. First function takes a regular value and wraps it inside your list. The function `<*>` can be read as ”apply” or ”app”, and it used to apply a wrapped function to a wrapped value.

```
instance Applicative MyList where
    pure = ...
    (<*>) = ...
```

```
pure 3 :: MyList Int ⇒ Cons 3 Nil Cons (+1) (Cons (*4) Nil) <*>
Cons 1 (Cons 2 Nil)
⇒ Cons 2 (Cons 3 (Cons 4 (Cons 8 Nil)))
```

(c) At last, it is time to implement Monad instance. This is easy, as the Applicative does most of the work.

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

You can notice that `return` has the same type signature as `pure`. Indeed, they are the same. The operator `(>>)` is defined in terms of the *bind* function, `(>>=)`. This means that you only need to define the *bind* function. A subset of type class functions that need to be implemented is called *minimal complete definition*. The Monad type class provides multiple functions, but its minimal complete definition is `(>>=)`.

```
  instance Monad MyList where
      (>>=) = ...
```

```
fun ::  Int -> MyList Int
fun x = Cons (x + 1) (Cons (x * 2) Nil)

Cons 1 (Cons 3 Nil) >>= fun ⇒ Cons 2 (Cons 2 (Cons 4 (Cons 6 Nil)))
Cons 1 (Cons 2 Nil) >> return 3 ⇒ Cons 3 (Cons 3 Nil)
```

You can read more about type classes on the Typeclassopedia.

Corrections

http://xkcd.com/1312/