U08074 Reasoning About Functional Programs Assignment

Part of this assignment is based on some exercises from the textbook *Haskell: the Craft of Functional Programming* by Simon Thompson, so you may find it useful to consult that book when doing the exercises.

Deadline: Wednesday 5pm Week 8 (23 March 2016).

Feedback: Prompt feedback by Wednesday 5pm Week 9 (13 April 2016).

Mark breakdown: Each question is worth 5 marks. Note that the marks for the answer include the marks for explanatory text and testing as well as the programs. So please submit your tests in your file. **Tests should be different from those supplied in the question.** For each question there are 3 marks for the function(s), 1 for explanation and 1 for tests. In some cases (e.g. questions 3, 4 and 7) you need to give a succinct answer (e.g. using higher-order functions) for full marks.

You should submit a single file of Haskell code. It would be best to submit a literate script (.lhs) file, rather than a .hs file, since then your tests and comments can be included more easily.

Please take advantage of the practical sessions to obtain **feedback** on your work at your own pace as you work through the tasks.

Part I : Supermarket billing

Given is a list of barcodes from the scanner at the checkout:

```
> type BarCode = Int
> type TillType = [BarCode]

> barcodes :: TillType
> barcodes = [1234, 4719, 3814, 1112, 1113, 1234]
```

and a database giving the barcode, name and price of every kind of item in the shop:

```
> type Name = String
> type Price = Int
> type Database = [ (BarCode, Name, Price) ]

> codeIndex :: Database
> codeIndex = [ (4719, "Fish Fingers" , 199),
>               (5643, "Nappies" , 1010),
>               (3814, "Orange Jelly", 89),
>               (1111, "Hula Hoops", 75),
>               (1112, "Hula Hoops (Giant)", 200),
>               (1234, "Dry Sherry, 1lt", 909)]
```

The objective of the program is to produce a nicely-formatted supermarket bill from these two pieces of information:

```
  Dry Sherry, 1lt...........9.09
  Fish Fingers..............1.99
  Orange Jelly..............0.89
  Hula Hoops (Giant)........2.00
  Unknown Item..............0.00
  Dry Sherry, 1lt...........9.09

  Total....................23.06
```

The program will involve two stages. The first stage converts the list of barcodes into a list of (Name,Price) pairs; the second stage formats this list of pairs nicely. We therefore introduce a name for the type of information that occurs between the two phases:

```
> type BillItem = (Name,Price)
> type BillType = [BillItem]
```

We will need two functions, for these two stages:

```
makeBill :: TillType -> BillType
formatBill :: BillType -> String
```

The whole program is formed by combining the two stages:

```
printBill :: TillType -> String
printBill tt = formatBill (makeBill tt)
```

(Actually, this will produce a string with the special characters, in particular the newline characters, unexpanded. To get a nice result we have to ask Haskell to evaluate

```
putStr (printBill barcodes)
```

instead of just `printBill barcodes`.)

Formatting

We address the second stage, the formatting phase, first.

Task 1

Given a number of pence, for example `1023`, the pounds and pence parts are given by ``1023 `div` 100`` and ``1023 `mod` 100``. Using this fact, and the `show` function, define a function

```
formatPence :: Int -> String
```

so that, for example,

```
formatPence 1023 = "10.23"
```

(Be careful about the case when the pence part is less than ten!)

Task 2

We introduce a constant for the length of the lines:

```
> lineLength :: Int
> lineLength = 30
```

Using the `formatPence` function, define a function

```
formatLine :: BillItem -> String
```

which formats a line of the bill, producing a line of length `lineLength`, thus:

```
formatLine ("Dry Sherry, 1lt", 540) = "Dry Sherry, 1lt...........5.40\n"
```

Recall that `'\n'` is the newline character, that `++` can be used to join two strings together, and that `length` will give the length of a string.

You may find it helpful to define a function

```
rep :: Int -> Char -> String
```

which builds a string containing a given number of occurrences of the same character. For example,

```
rep 4 'n' = "nnnn"
```

There is a Haskell function that will do this for you but you are encouraged to write the function from scratch.

Task 3

Define the function

```
formatLines :: BillType -> String
```

which applies `formatLine` to each `(Name,Price)` pair, and concatenates the results together.

Task 4

Define a function

```
makeTotal :: BillType -> Int
```

which takes a list of `(Name,Price)` pairs and gives the total of the prices. For instance,

```
makeTotal [ ("...",540), ("...",121) ] = 661
```

Task 5

Define the function

```
formatTotal :: Int -> String
```

so that, for example,

```
formatTotal 661 = "\nTotal..........6.61\n"
```

(Hint: use `formatLine`.)

Task 6

Finally, using the functions `formatLines`, `makeTotal` and `formatTotal`, define the function `formatBill`, so that on the input

```
[ ("Dry Sherry, 1lt",540), ("Fish Fingers",121),
  ("Orange Jelly",56), ("Hula Hoops (Giant)",133),
  ("Unknown Item",0), ("Dry Sherry, 1lt",540) ]
```

the example bill given earlier is produced.

This completes the definition of the formatting functions; now we have to look at the database functions which accomplish the conversion of barcodes into names and prices.

The database

Task 7

Define a function

```
look :: Database -> BarCode -> BillItem
```

which returns the `(Name,Price)` pair corresponding to the `BarCode` in the `Database`. If the `BarCode` does not appear in the database, then the pair `("Unknown Item",0)` should be the result. (You may assume that each barcode appears at most once in the database.)

Task 8

Define a function

```
lookUp :: BarCode -> BillItem
```

which uses `look` to look up an item in the particular database `codeIndex`.

Task 9

Hence define the function `makeBill` which applies `lookup` to every item in the list of barcodes. For instance, when applied to the list `[1234, 4719, 3814, 1112, 1113, 1234]`, the result will be the list of `(Name,Price)` pairs given in Exercise 4.42. Note that `1113` does not appear in `codeIndex`, and so appears as an unknown item.

This completes the definition of `makeBill`, which together with `formatBill` gives the complete program.

Task 10

Modify your script so that the bill is printed with the items sorted in ascending order by barcode. You should **not** use a library function to sort the list.

Part II: Dewey-decimal codes

This part of the assignment involves writing a program to manipulate library classification numbers, such as the Dewey-decimal classification system. Under such a classification system, an area of knowledge is subdivided into a hierarchy of sub-areas, sub-sub-areas and so on. Each little field is given a code, such that the code of a parent field is a prefix of the codes of the sub-fields within that field. For example, here is a (hypothetical) hierarchy of classifications and codes:

```
Code   Classification

0      Science
01     Physics
02     Computing
020    Computer hardware
021    Computer languages
0210   Imperative languages
0211   Functional languages
02110  Haskell
02111  ML
02112  Scala
022    Artificial Intelligence
03     Mathematics
1      Arts
2      Religion
```

(This is for a library with a distinct subject bias.)

We will represent (the names of) classifications by strings, and their codes by lists of symbols, where a symbol is merely a character.

```
> type Symbol = Char
> type Code = [Symbol]
> type Classfn = String
```

A hierarchical structure like this is of course very naturally represented by a tree of some kind. Each node in the structure represents a sub-field. At a node we store the symbol by which this sub-field extends the parent field's code, and the classification of that sub-field; we also store a list of children, one for each of the sub-sub-fields of this sub-field.

```
> data DeweyTree = Node Symbol Classfn DeweyForest
>   deriving Show

> type DeweyForest = [DeweyTree]
```

(The `deriving` clause allows us to use `show` to get a simple textual representation of one of these data structures.)

The whole hierarchy is modelled by a `DeweyForest` rather that a `DeweyTree`. For example, the hierarchy presented above is represented by the data structure

```
[ Node '0' "Science"
    [ Node '1' "Physics" [],
      Node '2' "Computing"
        [ Node '0' "Computer hardware" [],
          Node '1' "Computer languages"
            [ Node '0' "Imperative languages" [],
              Node '1' "Functional languages"
              [ Node '0' "Haskell" [],
                Node '1' "ML" [],
                Node '2' "Scala" []
              ]
            ],
          Node '2' "Artificial Intelligence" []
        ],
      Node '3' "Mathematics" []
    ],
  Node '1' "Arts" [],
  Node '2' "Religion" []
 ]
```

You will write functions to build such a structure, look up information in it, and print it out neatly.

Task 11

Define functions `symbol`, `classfn` and `children`, which extract the three components of a node.

```
> symbol :: DeweyTree -> Symbol

> classfn :: DeweyTree -> Classfn

> children :: DeweyTree -> DeweyForest
```

Task 12

Define functions `insertClassfnT` and `insertClassfn` to insert a new field (given its code and classification) into a tree or a forest. Your two functions will probably use each other, since the two types do too.

```
> insertClassfnT :: (Code,Classfn) -> DeweyTree -> DeweyTree

> insertClassfn :: (Code,Classfn) -> DeweyForest -> DeweyForest
```

You may assume that the classification to be inserted is not already present, but that the parent classification (whose code consists of all but the last symbol of the code given) *is* present. You may also assume that the given code is a non-empty list of symbols. (Hint: for inserting into a forest, there are four cases to consider. The code may consist of a single symbol or of more than one symbol, and the forest may be empty or non-empty.)

Task 13

Hence define a function to build a hierarchy from a whole list of fields (that is, from the codes and classifications).

```
> buildHierarchy :: [(Code,Classfn)] -> DeweyForest
```

You may assume that the fields are presented in a `top-down' order, so that the parent fields of a sub-field all appear earlier in the list that the sub-field itself, and that there are no duplicates in the list of field codes.

For example, given the following list of fields, `buildHierarchy info` should build the data structure presented above.

```
> info = [ ("0","Science"), ("1","Arts"), ("2","Religion"),
>          ("01","Physics"), ("02","Computing"), ("03", "Mathematics"),
>          ("020","Computer hardware"), ("021","Computer languages"), ("022","Artificial Intelligence"),
>          ("0210","Imperative languages"), ("0211","Functional languages"),
>          ("02110","Haskell"),("02111","ML"),("02112","Scala") ]
```

Task 14

Define a function to look up a particular code in the data structure, and return the corresponding classification.

```
> lookupClassfn :: Code -> DeweyForest -> Classfn
```

Task 15

Define a function to convert the data structure to a string, which when displayed using `putStr` produces a nicely-formatted hierarchy diagram.

```
> showForest :: DeweyForest -> String
```

For example,

```
? putStr (showForest (buildHierarchy info))
  0: Science
    01: Physics
    02: Computing
      020: Computer hardware
      021: Computer languages
        0210: Imperative languages
        0211: Functional languages
          02110: Haskell
          02111: ML
          02112: Scala
      022: Artificial Intelligence
    03: Mathematics
  1: Arts
  2: Religion
```

Clare Martin: cemartin@brookes.ac.uk