Functional and Logic Programming

Home Assignment 2

Due: Friday, 12.4.2019 - 23:55

Instructions

- Please create a source file called ***hw2.hs*** and put all the answers there.
  The file should start with a comment which contains your **full name** (in English) and **ID**

  ```
  -- John Doe
  -- 654321987
  ```

- Make sure the file **is valid** by loading it into GHCi.
  A valid file will load without any errors or warnings.
- If you need a function but you don't know how to implement it - just write it's signature (name and type) and put `undefined` in the function's body.
  That way you'll be able to load the file even though it contains references to undefined names. (
- When writing a function - write both the **type** and the **body** of the function.
- Be sure to write functions with **exactly the specified name** (and **type signature** - if it is provided) for each exercise.
  You may create additional auxiliary/helper functions with whatever names and type signatures you wish.
- Try to write **small functions** which perform just **a single task**, and then **combine** them to create more complex functions.

Exercises

1. Every function in Haskell takes a single argument and outputs a single value.
However, in Haskell, there are 2 ways to simulate a function which takes 2 arguments:

- Uncurried- a function which takes a tuple of 2 and executes the operation.
- Curried- A function which takes the first argument and outputs a function which takes the second argument and executes the operation on both arguments.

Each one has its own advantages and disadvantages.
We can easily convert an uncurried function to be curried and vice versa.
In this question you will implement 2 functions which perform this conversion.

Implement:

a. A function `myCurry` which takes an uncurried function and converts the function into a function of the second kind (curried):
`myCurry` **:: ((t1, t2) -> t) -> t1 -> t2 -> t**

b. A function `myUncurry` which performs the opposite:
`myUncurry` **:: (t1 -> t2 -> t) -> (t1, t2) -> t**

Examples:

```
*Main> :type (myCurry fst)
(myCurry fst) :: t -> t2 -> t
*Main> :type (myUncurry mod)
(myUncurry mod) :: Integral t => (t, t) -> t
```

2. Create a function:

```
bundler :: Eq a => [a] -> [[a]]
```

which bundles consecutive duplicates of list elements into sub-lists.
If a list contains repeated elements, they should be placed in separate sublists.
NOTE: You can assume you will be tested only on lists of Integers.

Examples:

```
bundler []=[]
bundler [1]=[[1]]
bundler [1,1,1,2,2,1,1,3] = [[1,1,1],[2,2],[1,1],[3]]
```

3. Create a function:

```
numAppearances :: Eq t => [t] -> [(Int, t)]
```

which takes a list of elements and outputs a list of pairs comprised the following way:
(number of consecutive duplicates of an element, an element).
If the given list contains repeated (nonconsecutive) elements, they should be placed in separate pairs.

Examples:

```
numAppearances [] = []
numAppearances ['a'] = [(1,'a')]
numAppearances ['a','a','a','b','a'] = [(3,'a'),(1,'b'),(1,'a')]
```

4. Create a function:

```
dropMod:: [a] -> Int -> [a]
```

which takes a list and an Int and drops every n^{th} element in the given list.
NOTE: You can assume $n > 0$

Examples:

```
dropMod [] 1 = []
dropMod [1,2,3] 1 = []
dropMod [1,2,3] 2 = [1,3]
dropMod [1,2,3,4,5,6] 3 = [1,2,4,5]
```

5. Implement a function `toBinary` which converts a decimal number into a binary number. (You can assume you will be tested on Integers)

```
toBinary:: Integral a => a -> [a]
```

Examples:

```
toBinary 0 = [0]
toBinary 1 = [0,1]
toBinary 3 = [0,1,1]
toBinary 4 = [0,1,0,0]
```

6. In this section we'll use a list of pairs to represent a simplified "database" where each item has a **non-unique** "key" (which is given as a **String**).
The database will support the following operations:

- `insertItem` - adds a (key,item) pair to the database.
- `itemsByKey` - get all the items with the given key.
- `getKeys` - get a list of all the keys.
- `groupItemsByKey` - group all items by their keys.

The database should be **polymorphic** - the implemented functions should work for **any** type of items (while the type of keys **must** be **String**).

a) Implement the `insertItem` function which takes a pair of **(**key,item**)** (where key is a **String** and item is **polymorphic**), and a list of **(**key,item**)** pairs - and adds the pair to beginning of the list.

```
insertItem :: (String, a) -> [(String,a)] -> [(String,a)]

insertItem ("a",8) [("",5),("x",2),("a",12)] = [("a",8),("",5),("x",2),("a",12)]
```

b) Implement the `itemsByKey` function which takes a key (given as a **String**), and a list of **(**key,item**)** pairs - and returns all the items in the list, which have the given key.

```
itemsByKey :: String -> [(String, a)] -> [a]

itemsByKey "a" [("a",8),("",5),("x",2),("a",12)] = [8,12]
```

c) Implement the `getKeys` function which takes a list xs of **(**key,item**)** pairs and returns a list of all the **keys** of xs, where each key appears **only once!**.
(**note:** the **order** of the elements in the result **doesn't matter!** - as long as all the keys are in the list and there are no duplicates)

```
getKeys :: [(String, a)] -> [String]

getKeys [("a",1),("b",2),("a",3),("a",4),("c",5),("b",6)] = ["a","b","c"]
```

d) Implement the `groupItemsByKey` function which takes a list xs of **(key,item)** pairs and returns a list of **(key,items)** pairs where each pair contains a key and a list of all the items in xs which have this key.
Each key must appear **only once** in the result.

(**note:** the **order** of the elements in the result **doesn't matter!**)

```
groupItemsByKey:: [(String, a)] -> [(String, [a])]


groupItemsByKey [("a",1),("b",2),("a",3),("a",4),("c",5),("b",6)] =
[("a",[1,3,4]),("b",[2,6]),("c",[5])]
```

7. Implement functions for each of the following types:

`bar` **:: (a -> (b -> c) -> c)-> ((a -> c) -> c) -> (b -> c) -> c**

`foo` **:: (a -> (b,c)) -> (b -> d) -> (c -> d -> e)-> a -> e**