Functional Programming for Logicians
Homework 6

Péter Mekis
Department of Logic, ELTE Budapest

Deadline: 2019 March 25 17:59 pm

Solve three of exercises 1-10, and three of exercises 11-20. Solving more is appreciated, but not necessary.

1–10 The `foldr` function for lists is defined as:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f y []     = y
foldr f y (x:xs) = foldr f (f x y) xs
```

Here's how it works:

```
foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...)
```

And a specific example:

```
foldr (^) 2 [3, 2, 1] == 3 ^ (2 ^ (1 ^ 2)) == 9
```

Use `foldr` to define the following functions. Do not use recursion or list comprehension.

sample `myElem' :: (Eq a) => a -> [a] -> Bool`
Eg. `myElem 'L' "Haskell" == False`

```
myElem :: (Eq a) => a -> [a] -> Bool
myElem z s = foldr (isit z) False s where
  isit :: (Eq a) => a -> a -> Bool -> Bool
  isit z x y = (x == z || y)
```

1. `myReverse :: [a] -> [a]`
   Eg. `myReverse "Haskell" == "lleksaH"`
2. `myLength :: [a] -> Int`
   Eg. `myLength "Haskell" == 7`
3. `mySum :: (Num a) => [a] -> a`
   Eg. `mySum [1,2,3] == 6`
4. `myProduct :: (Num a) => [a] -> a`
   Eg. `myProduct [1,2,3] == 6`
5. `myMaximum :: (Ord a) => [a] -> a`
   Eg. `myMaximum [False,True] == True`
6. `squareSum :: (Num a) => [a] -> a`
   Eg. `squareSum [1,2,3] == 14`

7. `factorial :: (Num a) => a -> a`
   Eg. `factorial 6 == 720`
8. `eraseItem :: (Eq a) => a -> [a] -> [a]`
   Eg. `eraseItem 'a' "Barack Obama" == "Brck Obm"`
9. `howMany :: (Eq a) => a -> [a] -> Int`
   Eg. `howMany 'a' "Barack Obama" == 4`
10. `parenthCheck :: String -> Bool`
    Eg. `parenthCheck "((2+3)*((4+5)/7))" == True`

11. In the session, we defined the `HunBool` type, deriving from a bunch of classes. Make `HunBool` an instance of `Ord`, `Enum`, and `Bounded` by means of explicite instance declarations, just as we did with `Eq` and `Show`.

12. Define a `Weekday` type with type constructors `Monday` ... `Sunday`. Make it an instance of the `Show`, `Read`, `Eq`, `Ord`, `Enum`, and `Bounded` classes.

13. In the session, we defined a `length` function for the `Tree` type. Define a `depth` function that will find the length of the longest branch of a tree. Eg. `depth(montagueTree) = 3`.

14. Define a function that checks whether a value of type `a` occurs as a label at a node or a leaf of a tree of type `Tree a`. Eg. `occurs "Bill" montagueTree == False`.

15. Define a function that flips a tree horizontally; eg. `treeFlip(montagueTree) == Node "S4" (Node "S5" (Leaf "Mary") (Leaf "love")) (Leaf "John")`

16. Define a `branches` function that will return all the branches of a tree, from root to leaf, as a list of lists. Eg. `branches(montagueTree) == [["S4","John"],["S4","S5","love"],["S4","S5","Mary"]]`

17. Redefine the `show` function for the `Tree` type so that it will show the structure of the tree with indentation. Use the `\n` character for line breaking, and call the `print` function to make line breaks visible.

```
> print montagueTree
"S4"
- "John"
- "S5"
- - "love"
- - "Mary"
```

18. Modify the `Tree` type so that the type of the data at the nodes may be different from the type of the data at the leaves; eg. there may be integers at the leaves, and arithmetic operations at the nodes. Define a few trees in the new type.

19. Another approach to binary trees is that a tree is either empty (constructor: `Empty`, no parameter), or it is a node with two branches. Define this version, and a few trees in this type.

20. Find a way to define a tree type with arbitrarily many branches at each node. Define a few trees in this type.