CS 3FP3: Functional Programming

Due on Thursday January 21rst, 2021

Dr. Jacques Carette

Idea

The goals of this assignment are:

1. get (re)started with Haskell

The Task

Do each of the following (but do pay attention to the submission requirements below)

1. Write the following 3 function

   ```
   matches :: Eq a => a -> [a] -> [a]
   element :: Eq a => a -> [a] -> Bool
   pos :: Eq a => a -> [a] -> [Integer]
   ```

   where matches returns a list of the elements from the given list that match (are equal to) the first argument (yes, this is silly); element returns a boolean which tells you if the first argument belongs to the list; and pos returns the *list* of 0-indexed positions where the first argument occurs.

   Answer (in comments) the following questions: which of the 3 functions, which do very related things, is "best"? Why do you believe that?

2. Write an "apply all" function of type

   ```
   applyAll :: [a -> b] -> [ a ] -> [ b ]
   ```

   which takes a list of function, and applies all of them to all the elements of the second list. For example, if the list of functions has 5 elements, and the list of values has 3, the resulting list has 15 elements.

3. Write functions with the following signature

   ```
   tripleNeg1 :: (Ord a, Num a) => [ a ] -> [ a ]
   tripleNeg2 :: (Ord a, Num a) => [ a ] -> [ a ]
   ```

   which goes over a list of numbers and triples each negative one. Write it twice,

   (a) using explicit recursion over the list

   (b) use the Haskell functions **either** and **map**

4. Create a new data type (call it OrBoth) that contains an a or a b or *both*. Then create functions with the following signature:

   ```
   consume1 :: (a -> c) -> (b -> c) -> (a -> b -> c) -> OrBoth a b -> c
   consume2 :: (a -> c) -> (b -> c) -> (c -> c -> c) -> OrBoth a b -> c
   ```

[Hint: look at the **either** function from the Prelude as a model]. The 'both' case for —consume2— must use *all* its arguments.

Further question to answer: which of those two ways is "better" in your opinion? Why?

5. Given the following definition for a *Ternary Tree*

```
data Ternary a = TLeaf a | TNode (Ternary a) (Ternary a) (Ternary a)
```

define two functions,

```
mirror :: Ternary a -> Ternary a
flattenTernary :: Ternary a -> [ a ]
```

which "mirror" (i.e. generalized reverse) the tree, and flattens the tree (where the left-most, deepest node ends up first, and right-most deepest node ends up last).

6. **First Steps To List Induction**

Given universal quantification over a list:

```
all :: (a -> Bool) -> [a] -> Bool
all p []     =  True
all p (x:xs) =  p x && all p xs
```

There is a "range split" rule which can be defined as follows:

$$\forall p, xs, ys. \texttt{all p (xs ++ ys)} = \texttt{all p xs} \wedge \texttt{all p ys}$$

Prove this by induction.

7. **Equational Specification** Given the specification

$$\forall xs, ys\ .\quad \texttt{mystery f xs ys} = \texttt{map f (zip xs yz)}$$

Produce an explicit definition of `mystery` that can only

- make recursive calls (to itself)
- make constructor calls
- apply `f`

Assume the least constraining amount of properties on `f` to carry out this task.

8. **Expressiveness of Fold**

"foldr" is the eliminator of the list data-type, which means every primitive recursive function on lists can be written as a fold. Express `reverse` as a fold.

Bonus: Do so by *calculating* the parameters of the required fold.

Hints:

- For some examples, look at the paper "A tutorial on the universality and expressiveness of fold", namely section 3.3, available at `http://www.cs.nott.ac.uk/~pszgmh/fold.pdf`
- By 'calculate', we mean to *start the proof* that the result is correct with some arbitrary `f` and `e` parameters to `foldr`, and derive some necessary and sufficient conditions on these.
- There will be a *generalization* step involved.

9. **Mirror, mirror on the wall** Consider the following code for a mirroring a binary tree:

```
data Tree a = Tip | Node (Tree a) a (Tree a)

mirrorTree :: Tree a -> Tree a
mirrorTree Tip = Tip
mirrorTree (Node l a r) = Node (mirrorTree r) a (mirrorTree l)
```

Define two functions `pre, post :: Tree a -> [a]` that traverse a tree and collect all stored values in *pre*-order and in *post*-order. Prove `pre (mirrorTree t) = reverse (post t)`.

10. **Trees, trees everywhere** Rose trees are, in fact, as expressive as binary trees. Here we show this constructively.

    We start with the type definitions

    ```
    data Rose a = Rose a [Rose a ]
    data Fork a = Leaf a | Branch (Fork a) (Fork a)
    ```

    and the `Tree` definition from above.

    Your task is to produce pairs of functions

    ```
    to'   :: Tree a -> List (Rose a)
    from' :: List (Rose a) -> Tree a

    to    :: Rose a -> Fork a
    from  :: Fork a -> Rose a
    ```

    The first two functions ought to show that `Tree a` is isomorphic to `List (Rose a)`; the second that `Rose` is isomorphic to `Fork`.

    Bonus: For the second isomorphism functions provide an inductive proof that ∀`xs. to (from xs) = xs`.

Submission Requirements

- Must contain at least a text file `A1_macid.lhs` (with your id, i.e. your email address, I am 'carette', substituted in) with all your Haskell code.
- The names of the file **does** matter.
- Code which **does not compile** is worth **0** marks.
- if you have looked things up online (or in a book) to help, document it in your code. If you have asked a friend for help, document that too.

Grading

- Haskell code: 80%
- English comments: 20%

Reminder: code that does not compile will make that whole part of the marking scheme be worth 0.

Notes

Questions that ask "what is better" will have marks associated to your reasoning, not to the plain answer. You may well get full marks if your reasoning is sound even though your opinion differs than ours.

For the functions themselves:

Q1 You may use pattern-matching, arithmetic, guards, if-then-else, `:`, `==`, `[]`, and list comprehension.

Q2 You may use pattern-matching, **map**, `++`, `:`, `$`, `[]` and `(.)` .

Q3 You may use pattern-matching, if-then-else, arithmetic, `:`, `[]` and list comprehension for tripleNeg1. You **must** use **map** and **either** for tripleNeg2 and whatever else you see fit.

Q4 You need to define a data type. Then you can use pattern-matching (and function composition).

Q5 You may use pattern-matching, `[]`, `:`, `++`, `$`, **concat**, and **map**, and list comprehension.

Q8 You must use **foldr**, and cannot use `revserse`. You may use pattern-matching, `++`, `[]` and `. .`

Note that you do not **have** to use all of them! Some are thrown in as distractors...

For questions not explicitly listed, they are either a proof, or you can use anything you want from the built-in libraries.

If you google hard enough, you'll be able to find answers to some of the questions above. If you have to do this, you're in big trouble, as you simply will not be able to do later assignments at all.

More Bonus

You should do the assignment itself, then the following on top of that.

1. Do the assignment using Org mode instead of using `.lhs`. Please hand in a tangled `.lhs` file as well as the `.org` file.
2. Do the assignment in Agda (or Idris). Including the properties – but these you should *prove*. Worth up to 10 final marks. Hand in either `.lagda` or `.lidr` files. Or both if you are super-keen.

Important note for Agda/Idris: you may wish to use `Vec` (sized lists) instead of lists many places lists are used above. It will, in some cases, make things simpler. And rather than the **Num** constraint, you should be parametric in the appropriate structure.