Programming in Haskell – Homework Assignment 5

UNIZG FER, 2016/2017

Handed out: December 9th, 2016. Due: December 15th, 2016 at 17:00

Note: Define each function with the exact name 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 ($\star$) 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 ($\star$) 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 some problems remaining unsolved.

1. *(2 pt)* Define a small domain-specific language for enslaving turtles. Using the data definitions below, implement the basic functions needed to control the turtle.

```
-- x and y coordinates
type Position = (Integer, Integer)
data Orientation = West | East | North | South deriving (Eq, Show)
-- Clockwise and Counterclockwise
data TurnDir = CW | CCW deriving (Eq, Show)
```

   (a) Define the data type Turtle in such a way to make solving the other subtasks possible.

   ```
   data Turtle = {...} deriving Show
   ```

   (b) Define the function `position` that takes a `Turtle` and returns its `Position`. (Hint: No need to define it explicitly. Recall record syntax).

   ```
   position :: Turtle -> Position
   ```

   (c) Define the function orientation that takes a Turtle and returns its Orientation. (Hint: Same as above)

   ```
   orientation :: Turtle -> Orientation
   ```

   (d) Define the function `newTurtle` that creates a `Turtle` at Position `(0, 0)` facing upwards.

   ```
   leonardo = newTurtle
   position leonardo ⇒ (0,0)
   orientation leonardo ⇒ North
   ```

(e) Define the function `move` that takes an `Integer` and a `Turtle` and moves the turtle a given amount in the direction it is currently facing.

```
move :: Integer -> Turtle -> Turtle
position $ move 20 leonardo ⇒ (0,20)
position $ move 0 leonardo ⇒ (0,0)
position $ move (-10) leonardo
⇒ error "Turtles cannot move backwards"
-- The above is not true
```

(f) Define the function `turn` that takes a `TurnDir` and a `Turtle` and changes the turtle's position accordingly.

```
turn :: TurnDir -> Turtle -> Turtle
orientation $ turn CCW leonardo ⇒ West
orientation $ turn CW $ turn CCW leonardo ⇒ North
position $ move 5 $ turn CW leonardo ⇒ (5,0)
position $ move 5 $ turn CCW $ move 10 $ turn CCW leonardo
⇒ (-10,-5)
```

(g) Implement a function `runTurtle` that enables us to chain our commands to the turtle more easily (Disclaimer: The name `slowWalkTurtle` would have been more apt, but `runTurtle` was chosen for brevity).

```
runTurtle :: [Turtle -> Turtle] -> Turtle -> Turtle
position $ runTurtle [move 5, turn CW, move 20, turn CW, move 10]
leonardo ⇒ (20,-5)

spiral i = move i : turn CW : spiral(i + 1)
position $ runTurtle (take 10 $ spiral 1) newTurtle ⇒ (-2,3)
```

2. *(2 pt)* Implement the following functions over a `Tree` data type defined as:

```
data Tree a = Leaf | Node a (Tree a) (Tree a) deriving (Eq, Show)
testTree = Node 1 (Node 2 Leaf Leaf) (Node 3 (Node 4 Leaf Leaf) Leaf)
```

(a) Define a function `treeFilter` that takes a predicate and a `Tree` and removes those subtrees that do not satisfy the given predicate (with any children).

```
treeFilter :: (a -> Bool) -> Tree a -> Tree a
treeFilter (const True) testTree ⇒ testTree
treeFilter odd testTree ⇒ Node 1 Leaf (Node 3 Leaf Leaf)
treeFilter (<3) testTree ⇒ Node 1 (Node 2 Leaf Leaf) Leaf
```

(b) Define a function `levelMap` that takes some binary function and applies it to the tree. The function that is being applied takes the depth of the tree as the first argument. The root element's depth is 0.

```
levelMap :: (Int -> a -> b) -> Tree a -> Tree b
levelMap (\l x -> if odd l then x else x + 1) testTree
⇒ Node 2 (Node 2 Leaf Leaf) (Node 3 (Node 5 Leaf Leaf) Leaf)
levelMap const testTree)
⇒ Node 0 (Node 1 Leaf Leaf) (Node 1 (Node 2 Leaf Leaf) Leaf)
```

(c) Define a function `isSubtree` that takes two instances of `Tree` and checks whether the first tree appears as part of the second.

```
isSubtree :: Eq a => Tree a -> Tree a -> Bool
isSubtree (Node 4 Leaf Leaf) testTree ⇒ True
```

```
isSubtree (Node 3 Leaf Leaf) testTree ⇒ False
isSubtree testTree testTree ⇒ True
isSubtree Leaf testTree ⇒ True
```

3. *(2 pt)* Define a recursive data type `Pred` that represents a boolean expression. Use the type constructors `And`, `Or` and `Not` to represent boolean operations and the `Val` constructor to represent a boolean value. Implement the `eval` function that takes a `Pred` and returns its evaluated `Bool` value.

```
expr = And (Or (Val True) (Not (Val True))) (Not (And (Val True)
           (Val False)))

eval (Val True) ⇒ True
eval (Val False) ⇒ False
eval (Or (Val True) (Val False)) ⇒ True
eval expr ⇒ True
eval (Not expr) ⇒ False
```

4. *(2 pts)*

   (a) Someone was a messy music album uploader and uploaded list of track names formatted as: `"TrackTitle TrackNo AlbumName"`. You want from your music player to play those tracks sorted by track number, so write a function `sortTracks` that does that using higher-order functions.

   ```
   sortTracks :: [String] -> [String]
   sortTracks ["Different 02 In Silico","Propane Nightmares 03 In
   Silico","Showdown 01 In Silico","Visions 04 In Silico"]
   ⇒ ["Showdown 01 In Silico","Different 02 In Silico","Propane
   Nightmares 03 In Silico","Visions 04 In Silico"]
   ```

   (b) Music player prepended the number of times each track was played to the track name. Write a function `numberOfPlays` that calculates the number of played tracks for a whole album (use `fold`).

   ```
   numberOfPlays :: [String] -> Integer
   numberOfPlays ["10 Different 02 In Silico","16 Propane Nightmares
   03 In Silico","14 Showdown 01 In Silico","9 Visions 04 In Silico"]
   ⇒ 49
   ```

5. *(⋆) (2 pts)* Stephen just moved into Zagreb and he is having a bit of trouble getting by. Every time he gets lost, he calls his good friend John and sends him a text message containing his current location. John then replies with a message explaining step-by-step which way Stephen should go so they can meet. John's reply consists of words `N, NE, E, SE, S, SW, W, or NW` separated by commas which represent the directions in which Stephen should step. For example the message `"N, N, S, S, S, N"` tells Stephen to make two steps North, three steps south, and then again one step North. Unfortunately, since John is frequently busy, he sometimes doesn't have enough time to type in the commas, which results in ambiguous messages. This made Stephen wonder, how many possible routes are there in a message that doesn't have commas? Examples:

```
possibleRoutes ::  String -> Int
possibleRoutes "NNNSSS" ⇒ 1
```
Since the only possible route is "N, N, N, S, S, S".
```
possibleRoutes "NEWS" ⇒ 2
```
"NE, W, S" and "N, E, W, S" are possible.
```
possibleRoutes "NESW" ⇒ 4
```
"N, E, S, W", "N, E, SW", "NE, S, W", "NE, SW" are possible.

Disclaimer: jk, we actually like turtles.

Corrections

Correction 1: The data constructors in the first task were causing conflicts with Prelude's Left and Right implementations. They directions were renamed Left → West, Right → East, Up → North, Down → South. Fixed a typo in the first task `moven` → `move`

Correction 2: Fixed a mistake in the second example for `turn` function. Replaced the result `(-10, 5)` with `(-10, -5)`

"We find your lack of reviews disturbing."
sayeth the students unto the TAs