CPSC 449 – Programming Paradigms

Assignment #3

Due Date: Friday, November 09, 11:59 PM

Individual Work:

All assignments in this course are individual work. Students are advised to read the guidelines for avoiding plagiarism located on the course outline. Students are also advised that electronic tools may be used to detect plagiarism.

Late Penalty:

Late submissions will not be accepted.

Submissions:

Submit `assignment03.hs` file that contains the required functions in `D2L`.

Problem Description:

1- Define a type `Season` that has the values `Fall`, `Winter`, `Summer` and `Spring`. Define another type for `Month` that has name of the month as a value such as `January`, `February`, `March` and so on.

Define a function:

```
months :: Season -> (Month, Month, Month)
```

that returns the months for that season:

- Winter is January, February and March
- Spring is April, May and June
- Summer is July, Aug and September
- Fall is October, November, and December

Sample Run:

```
*Main> months Fall
(October,November,December)

*Main> months Winter
(January,February,March)
```

```
*Main> months Summer
(July,August,September)

*Main> months Spring
(April,May,June)
```

2- Consider the following recursive type, modeling logical formulas in Haskell

```
data Form = And Form Form | Or Form Form | Not Form | Val Bool
```

For example, the formula `not True || False` is represented as `Or (Not (Val True)) (Val False)`.

Define a function:

```
eval :: Form -> Bool
```

That evaluates a given formula to its value, a Boolean.

Sample Run:

```
*Main> eval (And (Val True) (Or (Val False) (Val True)))

True

*Main> eval (Not (And (Val True) (Val False)))

True
```

3- Consider the following recursive data type

```
data NTree = Leaf Int | Node NTree Int NTree
```

Define a function:

```
collapse :: Tree -> [Int]
```

that returns all integers in a given tree as a list.

Sample Run:

```
*Main> collapse ( Node (Node (Leaf 1) 3 (Leaf 4)) 5 (Node
(Leaf 6) 7 (Leaf 9)))

[1,3,4,5,6,7,9]
```

4- Consider the following recursive data type

```
data PTree a = PLeaf | PNode a (PTree a) (PTree a)
              deriving Show
```

Define a function:

```
countLeaves :: PTree a -> Integer
```

that returns the number of leaves in a tree.

Sample Run:

```
*Main> countLeaves (PNode 1 (PNode 2 (PNode 3 PLeaf PLeaf)
PLeaf) (PNode 4 PLeaf PLeaf))

5

*Main>  countLeaves  (PNode  "Calgary"  (PNode  "Edmonton"
(PNode  "RedDeer"  PLeaf  PLeaf)  PLeaf)  (PNode  "Lethbridge"
PLeaf PLeaf))

5
```

5- Consider the following recursive data type, used to store integers in a tree shape.

```
data Store = Empty | Join Int Store Store
```

Define a function:

```
maxStore :: Store  ->  Int
```

that finds the largest integer element in the tree.

Sample Run:

```
*Main> maxStore Empty

0
```

```
*Main> maxStore (Join 8 Empty (Join 20 Empty Empty))

20
```

6- Given the following data type for arithmetic expression:

```
data Expr = Num Integer | BinOp Op Expr Expr
  deriving (Eq,Show)

data Op = Add | Mul
  deriving (Eq,Show)
```

Define a function:

```
countOp :: Op -> Expr -> Int
```

which counts the number of occurrences of a given operator in an expression. For example, the expression representing `1 + 2 + 3 * 4` is

BinOp Add (Num 1) (BinOp Add (Num 2) (BinOp Mul (Num 3) (Num 4)))

Sample Run:

```
*Main> example = BinOp Add (Num 1)(BinOp Add (Num 2)(BinOp
Mul (Num 3) (Num 4)))

*Main> countOp Add example
2

*Main> countOp Mul example
1
```

7- Consider a following type

```
data Tree a = Nil | Value a (Tree a) (Tree a)
          deriving Show
```

Define a following function:

```
mapTree :: (a -> b) -> Tree a -> Tree b
```

In order to test the solution, you should define a function

```
countChars :: String -> Integer
```

which returns number of characters in a string (Hint: toInteger may be useful)

Sample Run:

```
*Main> a = (Value "foo" (Value "bar" (Value "baz" Nil Nil)
Nil)  (Value "bam" Nil Nil))

*Main> mapTree countChars a
Value 3 (Value 3 (Value 3 Nil Nil) Nil) (Value 3 Nil Nil)

*Main> result = mapTree countChars (Value "78" (Value "br"
(Value "bz" Nil Nil) Nil)  (Value "am" Nil Nil))

*Main> print result
Value 2 (Value 2 (Value 2 Nil Nil) Nil) (Value 2 Nil Nil)
```

8- For this question, the Tree definition would be same that you have defined in the previous questions.

Define a function foldTree

```
foldTree :: (a -> a -> a) -> a -> Tree a -> a
```

Now, use this `foldTree` and `mapTree` to count the characters in a given Tree of string

Sample Run:

```
*Main> tree = (Value "78" (Value "br" (Value "bz" Nil Nil)
Nil) (Value "am" Nil Nil))

*Main> foldTree (+) 0 (mapTree countChars tree)
8
```

9- Consider the following recursive data type that represents road.

```
data Road = City String  | Fork Road Road

     deriving (Show)
```

`City c` represents a road that goes straight to the city c.

`Fork l r` represents a road on which turning left leads to another road l and turning right leads to the road r.

Here is an example of road:

```
middleOfNowhere = Fork ( City "Banff" ) ( Fork ( Fork ( City
"Edmonton" ) ( City "RedDeer" ) ) ( City "Calgary" ))
```

Define a function:

```
reachable :: String -> Road -> Bool
```

that checks whether a given city is reachable from the given road.

Hints:

Since every city in a Road can be reached from the root, checking whether a city is reachable is the same as checking whether it is included in the Road.

Sample Run:

```
*Main> middleOfNowhere = Fork ( City "Banff" ) ( Fork (
Fork ( City "Edmonton" ) ( City "RedDeer" ) ) ( City
"Calgary" ))

*Main> reachable "Lethbridge" middleOfNowhere
False

*Main> reachable "Edmonton" middleOfNowhere
True

*Main> reachable "" middleOfNowhere
False
```

10- In the previous questions, you saw a data type for representing roads. When a new road is built around a city, the map of the roads needs to be updated.

Define a function:

```
insertRoad :: (Road,LR) -> String -> Road -> Road
```

which inserts a new road before a city.

A call to this function takes the form `insertRoad (new,lr) city old`, where new is the new road to be inserted, `lr` determines whether the new road is entered by a left turn or a right turn (the `LR` type is defined below), city gives the name of the city before which the road should be inserted, and old is the original map of roads.

The LR type represents left or right turns, and is defined as follows:

```
data LR = L | R
```

Sample Run:

```
*Main> middleOfNowhere = Fork ( City "Banff" ) ( Fork (
Fork ( City "Edmonton" ) ( City "RedDeer" ) ) ( City
"Calgary" ))

*Main> test_insertRoad = insertRoad (City "Kananaskas",L)
"Calgary" middleOfNowhere

*Main> print test_insertRoad
Fork (City "Banff") (Fork (Fork (City "Edmonton") (City
"RedDeer")) (Fork (City "Kananaskas") (City "Calgary")))
```

Additional Challenge:

Alpha has made a car that accepts only four instructions: forward, backward, turn left and turn right. When the car turns right or left, it always turns 90 degrees. Alpha is not good in driving and in order to improve his driving skills, Alpha wants to write a computer simulator for the car's movement.

Alpha starts by modeling the four instructions as a data type

```
data Instruction = FORW Int | BACKW Int | LEFT | RIGHT
```

The integer argument to FORW and BACKW denotes the distance the car should drive in that direction.

Now Alpha is stuck and wants your help for car's simulator.

Define a function:

```
destination :: [Instruction] -> (Int,Int)
```

that, given a list of instructions computes the position of the car after following these instructions. The original position of the car is $(x, y) = (0, 0)$, and it is facing "upwards" in the sense that going forwards will increase its y position.

Hints:

You may find it useful to define a data type direction (North, South, East and West) and some helper functions e.g `turnLeft` that change the direction.

Sample Run:

```
*Main> destination [FORW 20, BACKW 10, RIGHT, FORW 100]
(100,10)

*Main> destination [FORW 20, BACKW 5, LEFT, FORW 100]
(-100,15)
```

Grading:

A+: All functions (1 – 10) and additional challenge of the assignment are completed successfully and pass all the test cases.

A: All functions (1 - 10) except for additional challenge are completed successfully.

A-: Any 9 functions are completed successfully.

B+: Any 8 functions are completed successfully.

B: Any 6 - 7 functions are completed successfully.

B-: Any 5 functions are completed successfully.

C: Any 4 functions are implemented successfully.

D / F: Submissions that do not meet the standard for a C will be awarded a grade of D+, D or F depending on functionality, quality and quantity of the submitted code.

Credits:

The text in this assignment is adapted from the two courses on Haskell offered at California Institute of Technology (Caltech) and Chalmers Institute of Technology.