MATHFUN
Discrete Mathematics and Functional Programming

Worksheet 7: Algebraic Types

Introduction

This worksheet aims to give you practice in defining algebraic types and functions that compute over them. Begin by downloading the Week7.hs file from the unit web-site which includes some definitions from the lecture - experiment with these definitions before moving onto the exercises.

Enumerated types

1. Define two algebraic types:

   - `Month` to represent the twelve months of the year;
   - `Season` to represent the four seasons.

2. Define a function:

   ```
   season :: Month -> Season
   ```

   which maps months onto seasons (assume `season February = Winter`, `season March = Spring` and that seasons are all three months long.). Try to make your definition as short as possible.

3. Define a function:

   ```
   numberOfdays :: Month -> Int -> Int
   ```

   which gives the number of days a month has in a given year. Assume all years divisible by four are leap years. For example, `numberOfDays February 2012 = 29`.

Points and Shapes

4. Define an algebraic type `Point` for representing (the coordinates of) points in two-dimensional space.

5. Using `Point`, define a modified version `PositionedShape` of the `Shape` data type which includes the centre point of a shape, in addition to its dimensions.

6. Define a function:

   ```
   move :: PositionedShape -> Float -> Float -> PositionedShape
   ```

   which moves a shape by the given x and y distances.

Functions for the binary tree type

7. Define, for the binary tree type `Tree`, a function:

   ```
   numberOfNodes :: Tree -> Int
   ```

   which returns the number of nodes there are in a given binary tree.

8. Define a function:

```
isMember :: Int -> Tree -> Bool
```

which tests whether a given value is in a tree.

9. Define a function:

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

which gives a list of the leaves of the tree (i.e. those nodes with null left and right subtrees).

10. Define a function:

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

which lists the elements of a tree according to an **in-order** traversal. (If the tree is a valid **binary search tree**, then this function will give a list of the tree's elements in ascending numerical order.)

11. Define a function:

```
insert :: Int -> Tree -> Tree
```

which inserts a new value into a tree. The function should assume that the tree is a binary search tree and should preserve this property.

12. [harder] Define a function:

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

which creates a binary search tree by inserting into an initially empty tree the elements from a list in the order in which they appear. For example:

```
listToSearchTree [2,1,3] =
            Node 2 (Node 1 Null Null) (Node 3 Null Null)
```

Finally, using `listToSearchTree` and `inOrder`, write another function:

```
binaryTreeSort :: [Int] -> [Int]
```

that sorts a list of integers.