CS 421 — Algebraic Data Type Activity

Mattox Beckman

Tuples

1. Can you write a function `tupLen` that takes a tuple and returns how many parts it has? For example:

```
Prelude> tupLen (1,5)
2
Prelude> tupLen (4,3,6,5)
4
```

2. Write the function `assoc` that takes a key, a value, and an associative list, and inserts the key-value pair into the list, preserving the property that the list is sorted by key.

```
Prelude> :t assoc
assoc :: Ord a => a -> t -> [(a, t)] -> [(a, t)]
Prelude> assoc "Jenni" 8675309 [ ("Emergency",911), ("Empire",5882300)]
[("Emergency",911),("Empire",5882300),("Jenni",8675309)]
```

3. Write a function `get` that takes a pair of pairs and two integers and traverses the pairs to find an element. 0 means "go left", and 1 means "go right".

   For example, `get 1 0` means take the left element of the right pair, as below.

```
Prelude> :t get
get :: (Eq a, Eq a1, Num a, Num a1) => a -> a1 -> ((t, t), (t, t)) -> t
Prelude> get 1 0 ((2,4),(5,6))
5
```

4. Now create a data type `Direction` that has two members `GoLeft` and `GoRight`. Rewrite `get` to use these.

Maybe

5. Write a function `maybePlus` that adds two `Maybe` types.

```
Prelude> maybePlus Nothing (Just 3)
Nothing
Prelude> maybePlus (Just 10) (Just 22)
Just 32
```

6. Write a function `maybeMap` that takes a maybe and a list and maps the function in the maybe to the list (or else just returns the list.)

```
Prelude> maybeMap (Just (+1)) [1,2,3]
[2,3,4]
Prelude> maybeMap Nothing [1,2,3]
[1,2,3]
```

7. Write a function `lift` that takes an operator and returns a new one that works with maybes the way `maybePlus` does.

```
Prelude> let maybeTimes = lift (*)
Prelude> maybeTimes Nothing (Just 4)
Nothing
Prelude> maybeTimes (Just 4) (Just 3)
Just 12
```

Trees

Here is a data type to implement a binary tree.

```
data Tree a = Node a (Tree a) (Tree a)
            | Empty
     deriving Show

add_bst :: Integer -> Tree Integer -> Tree Integer
add_bst i Empty = Node i Empty Empty
add_bst i (Node x left right)
   | i <= x    = Node x (add_bst i left) right
   | otherwise = Node x left (add_bst i right)
```

8. Write the function `del :: a -> Tree a -> Tree a` that will delete an element according to the binary search tree protocol.

9. Write a function `list2tree` that will create a tree out of all the elements of the list.

   (a) Can you do it so that the first element of the list is the root?

   (b) Can you do it using higher order functions?

10. Write a function `isBST` that takes a tree and determines if it is in fact a binary search tree or not. The best solution will run in $\mathcal{O}(n)$.