IN3043 Functional Programming
Solutions to Exercises 8

1. Some values of the type for use in testing:

```
example1 :: LTree Int
example1 = Branch (Leaf 2) (Leaf 3)

example2 :: LTree Int
example2 = Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3)
```

   (a) The number of leaves in a tree:

```
size :: LTree a -> Int
size (Leaf x) = 1
size (Branch l r) = size l + Size r
```

   (b) The depth of a tree:

```
depth :: LTree a -> Int
depth (Leaf x) = 0
depth (Branch l r) = (depth l `max` depth r) + 1
```

   (c) The list of leaf values:

```
leaves :: LTree a -> [a]
leaves (Leaf x) = [x]
leaves (Branch l r) = leaves l ++ leaves r
```

   (d) A leaf tree can be reversed by swapping the arguments of each **`Branch`**:

```
revLTree :: LTree a -> LTree a
revLTree (Leaf x) = Leaf x
revLTree (Branch l r) = Branch (revLTree r) (revLTree
   l)
```

2. (a) Replacing **`(+)`** in **`sumLTree`** with a parameter function f:

```
foldLTree :: (a -> a -> a) -> LTree a -> a
foldLTree f (Leaf x) = x
foldLTree f (Branch l r) = f (foldLTree f l)
   (foldLTree f r)
```

(b) **sumLTree** can be redefined as

```
sumLTree = foldLTree (+)
```

(c) If g replaces each leaf value with the constant **1**, then summing those will give the number of leaves. We can express that function as `\ x -> 1`, so we have

```
size = sumLTree . mapLTree (\ x -> 1)
```

The standard function

Prelude
```
const :: a -> b -> a
```

makes constant functions, so we can also use `const 1`:

```
size = sumLTree . mapLTree (const 1)
```

(d) We can redefine our functions as

```
size = foldLTree (+) . mapLTree (\ x -> 1)

depth = foldLTree (\ n m -> max n m + 1) . mapLTree (\
   x -> 0)

leaves = foldLTree (++) . mapLTree (\ x -> [x])

revLTree = foldLTree (\ l r -> Branch r l) . mapLTree
   Leaf
```

Some of these can be simplified using the standard functions **const** and **flip**:

```
size = foldLTree (+) . mapLTree (const 1)

depth = foldLTree (\n m -> max n m + 1) . mapLTree
   (const 0)

revLTree = foldLTree (flip Branch) . mapLTree Leaf
```

3. We have mutually defined types **Element** and **Content**, so it is appropriate to use mutually defined functions:

```
printElement :: Element -> String
printElement (Element n attrs []) =
    "<" ++ unwords (n : map printAttr attrs) ++ "/>"
printElement (Element n attrs body) =
    "<" ++ unwords (n : map printAttr attrs) ++ ">" ++
    concat (map printContent body) ++
    "</" ++ n ++ ">"
```

```
printContent :: Content -> String
printContent (Text s) = encodeString s
printContent (Child e) = printElement e
```

The function **printAttr** converts a single attribute-value pair to a string:

```
printAttr :: Attribute -> String
printAttr (n, val) = n ++ "=\"" ++ encodeString val ++
   "\""
```

Encoding of strings encodes each character:

```
encodeString :: String -> String
encodeString = concat . map encodeChar

encodeChar :: Char -> String
encodeChar '<' = "&lt;"
encodeChar '>' = "&gt;"
encodeChar '&' = "&amp;"
encodeChar '"' = "&quot;"
encodeChar c
  | isAscii c = [c]
  | otherwise = "&#" ++ show (ord c) ++ ";"
```

This version does a little more encoding than necessary, *e.g.* a double quote is safe outside an attribute value, though this is harmless.

4. We need to add two more constructors to the **Prop** type:

```
    | Or (Prop a) (Prop a)
    | Equiv (Prop a) (Prop a)
```

Then we need to add equations to handle the new constructors in all of the functions on this type. For example, in **mapProp**, we add

```
mapProp f (Or p q) =
    Or (mapProp f p) (mapProp f q)
mapProp f (Equiv p q) =
    Equiv (mapProp f p) (mapProp f q)
```

and in **evalBool** we add

```
evalBool (Or p q) = evalBool p || evalBool q
evalBool (Equiv p q) = evalBool p == evalBool q
```

5. A recursive version:

```
paths :: LTree Bool -> [Path]
paths (Leaf b)
  | b = [[]]
  | otherwise = []
paths (Branch l r) = map (L:) (paths l) ++ map (R:)
    (paths r)
```

This function can also be written using out higher-order functions:

```
paths :: LTree Bool -> [Path]
paths = foldLTree branch . mapLTree leaf
  where
    leaf b = if b then [[]] else []
    branch lps rps = map (L:) lps ++ map (R:) rps
```

6. (a) This function is similar in structure to **`vars`**:

```
foldProp f (Var v) = v
foldProp f (Not p) = foldProp f p
foldProp f (And p q) =
     f (foldProp f p) (foldProp f q)
foldProp f (Or p q) =
     f (foldProp f p) (foldProp f q)
foldProp f (Imply p q) =
     f (foldProp f p) (foldProp f q)
foldProp f (Equiv p q) =
     f (foldProp f p) (foldProp f q)
```

(b) We first replace each variable by the constant **`1`**, and then add all of these:

```
varCount :: Prop a -> Int
varCount p = foldProp (+) (mapProp (\ _ -> 1) p)
```

Using **`const`**, our function becomes

```
varCount :: Prop a -> Int
varCount p = foldProp (+) (mapProp (const 1) p)
```

or equivalently

```
varCount :: Prop a -> Int
varCount = foldProp (+) . mapProp (const 1)
```

(c) We use **`mapProp`** to replace each variable with a singleton set, and then use **`foldProp`** to form the union of those:

```
vars :: Ord a => Prop a -> Set a
vars p = foldProp Set.union (mapProp Set.singleton p)
```

or equivalently

```
vars :: Ord a => Prop a -> Set a
vars = foldProp Set.union . mapProp Set.singleton
```