Exercises
Declaring Types, Trees

Exercise 1:
Looking at the following definition of *tail* :

```
tail :: [a] -> [a]
tail [] = []
tail (_:xs) = xs
```

Rewrite this (call it *safetail*) using:

1. (*safetailMaybe*) Maybe (returning Nothing when called on empty list)
2. (*safetail"*) Either (returning error message when called on empty list)

Solution 1:

1.
```
safetailMaybe :: [a] -> Maybe [a]
safetailMaybe [] = Nothing
safetailMaybe xs = Just (tail xs )
```

2.
```
safetailEither :: Eq a => [a] -> Either String
  [a]
safetailEither xs = if null xs then Left "Cannot have tail of empty
                                    else Right (tail xs)
```

Exercise 2:
Referring to the abstract machine written in class notes:

```
-- Haskell Code for Abstract Machine example, Hutton, Chapter 7/
data Expr = Val Int | Add Expr Expr | Mult Expr Expr
data Op = EVAL Expr | ADD Int | MULT Int

type Cont = [Op]

eval :: Expr -> Cont -> Int
eval (Val n)    c = exec c n
eval (Add x y)  c = eval x (EVAL y: c)

exec :: Cont -> Int -> Int
exec []           n = n
```

```
exec (EVAL y: c)      n = eval y (ADD n: c)
exec (ADD n : c)      m = exec c (n + m)

val :: Expr -> Int
val e = eval e []
```

Write out the evaluation of the following Expressions

1. `Val 1`
2. `Add (Val 1) (Val 2)`
3. `(Add (Add (Val 2) (Val 3) ) (Val 4))`

Solution 2:

1. **Val 1**

```
value Val 1 = eval Val 1 []
            = exec [] 1
            = 1
```

2. **Add (Val 1) (Val 2)**

```
value (Add (Val 1) (Val 2))
            = eval (Add (Val 1) (Val 2))  []
            = eval (Val 1) (EVAL (Val 2) : [])
            = eval (Val 1) [EVAL (Val 2)]  --prepend to an empty list
            = exec [EVAL (Val 2)]  1
            = eval (Val 2) [ADD 1]
            = exec [ADD 1] 2
            = exec [] (1 + 2)
            = 3
```

3. **(Add (Add (Val 2) (Val 3)) (Val 4))**

```
value (Add (Add (Val 2) (Val 3) ) (Val 4))
            = eval (Add (Add (Val 2) (Val 3) ) (Val 4)) []
            = eval (Add (Val 2) (Val 3) [EVAL Val 4]
            = eval Val 2   [EVAL Val 3, EVAL Val 4]
            = exec [EVAL Val 3, EVAL Val 4] 2
            = eval Val 3 [ADD 2, EVAL Val 4]
            = exec [ADD 2, EVAL Val 4] 3
            = exec [EVAL Val 4] (2 + 3)
            = exec [EVAL Val 4]  5
            = eval Val 4 [ADD 5]
            = exec [] (5 + 4)
            = 9
```

Exercise 3:

The abstract machine (as per above) only implements ***additon*** . Show how you would extend this implementation to implement mutliplication.

Solution 3:

The abstract machine is extended as follows

```
data Expr = Val Int | Add Expr Expr | Mult Expr Expr
data Op = EVALA Expr | EVALM Expr | ADD Int | MULT Int

type Cont = [Op]

eval :: Expr -> Cont -> Int
eval (Val n)    c = exec c n
eval (Add x y)  c = eval x (EVALA y: c)
eval (Mult x y) c = eval x (EVALM y: c)

exec :: Cont -> Int -> Int
exec []          n = n
exec (EVALA y: c)  n = eval y (ADD n: c)
exec (EVALM y: c)  n = eval y (MULT n:c)
exec (ADD n : c)   m = exec c (n + m)
exec (MULT n : c)  m = exec c (n * m)

val :: Expr -> Int
val e = eval e []
```

Exercise 4:

(Using the Nat example from earlier)

In a similar manner to the function ***add***, define a recursive multiplication function

```
mult :: Nat -> Nat -> Nat
```

for the recursive type of natural numbers.

Hint: Make use of ***add*** in your definition **Solution 4:**

```
mult m Zero     = Zero
mult m (Succ n) = add m (mult m n)
```

Exercise 5:

Using the following (as seen in class) :

```
data Ordering = LT | EQ | GT
```

together with a function

```haskell
compare :: Ord a => a -> a -> Ordering
```

that decides if one value if an ordered type is less than (LT), equal to (EQ), or greater than (GT) another value. Using this function, redefine the function

```haskell
occurs :: Ord a => a -> Tree a -> Bool
```

for search trees. Why is this new definition more efficient that the original version? **Solution 5:**

```haskell
occurs x (Leaf y)     = x == y
occurs x (Node l y r) = case compare x y of
                          LT -> occurs x l
                          EQ -> True
                          GT -> occurs x r
```

This version is more efficient because it only requires one comparison between x and y for each node, whereas the previous version may require two.

Exercise 6:

Consider the following type of binary trees:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Let us say that such a tree is *balanced* if the number of leaves in the left and right subtree differs by at most one, with the leaves themselves being trivially balanced.

1. Define a function ***leaves*** that returns the number of leaves in a tree.

   ```
   leaves:: Tree a -> Int
   ```

2. Using ***leaves*** above, or otherwise, define a function ***balanced*** that decides if a tree is balanced or not.

   ```
   balanced:: Tree a -> Bool
   ```

3. Define a function ***depth*** that calculates the depth of a tree, where the depth is given by the number of nodes in the longest path from the root of the tree to a leaf in the tree.

   ```
   depth :: Tree a -> Int
   ```

Solution 6:

```
leaves (Leaf _)     = 1
leaves (Node l r) = leaves l + leaves r

balanced (Leaf _)     = True
balanced (Node l r) = abs (leaves l - leaves r) <= 1
                            && balanced l && balanced r

depth (Leaf _) = 0
depth ( Node l r ) = max (1 + depth l) 1 + depth r
```

Exercise 7:

Define a function

```
balance :: [a] -> Tree a
```

that converts a non-empty list into a balanced tree.
Hint: first define a function that splits a list into two halves whose length differs by at most one.

Solution 7:

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
              deriving (Show, Read)   -- so we can see it working

halve :: [a] -> ([a], [a])
halve xs = splitAt (length xs `div` 2) xs

balance ::  [a] -> Tree a
balance [x] = Leaf x
balance xs = Node ( balance ys ) ( balance zs)
             where (ys,zs) = halve xs
```

Exercise 8:

Using the idea of the search tree used in class with a slight change,

```
data Tree a =
             EmptyTree
           | Node (Tree a) a (Tree a)  deriving (Show, Read, Eq)

occurs x (Node l y r) | x == y = True
                      | x < y  = occurs x l
                      | x > y    = occurs x r

treeInsert :: (Ord a) => a -> Tree a -> Tree a
--Write the code for this

flatten :: Tree a ->  [a]
-- Write the code for this
```

Solution 8:

```
data Tree a =
             EmptyTree
           | Node (Tree a) a (Tree a)  deriving (Show, Read, Eq)

occurs x (Node l y r) | x == y = True
                      | x < y  = occurs x l
                      | x > y    = occurs x r

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = Node  EmptyTree x EmptyTree

treeInsert x (Node left a right)
 --    | x == a = trace ( "Equals" ) Node left x  right
    | x <= a  = trace ( "x < a" ) Node  (treeInsert x left) a right
    | x > a  = trace ( "x > a" )Node  left  a (treeInsert x right)

flatten :: Tree a ->  [a]
flatten EmptyTree    = []
flatten (Node EmptyTree x EmptyTree)      = [x]
flatten (Node l x r) = flatten l
                     ++ [x]
                     ++ flatten r
```

Exercise 9:

Define appropriate versions of the library functions:

1. ```
   repeat :: a -> [a]
   repeat x = xs where xs = x:xs
   ```

2. ```
   take :: Int -> [a] -> [a]
   take  0  _          = []
   take  _  []     = []
   take n (x:xs)   = x : take (n - 1) xs
   ```

3. ```
   replicate :: Int -> a -> [a]
   replicate  n = take n . repeat
   ```

4. ```
   map : (a-> b) -> [a] -> [b]
   map _ [] = []
   map f (x:xs) = f x : map xs
   ```

for the following type of binary trees:

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

You should test it on the following test Tree (or similar)

```
myTree :: Tree Int
myTree = Node  1 (Node 6 (Node 4 (Leaf)  (Leaf))  (Leaf) )
(Node 3 Leaf  Leaf )
```

Solution 9:

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

repeatTree :: a -> Tree a
repeatTree x =  Node x t t
                  where t = repeatTree x
takeTree :: Int -> Tree a -> Tree a
takeTree 0 _ = Leaf
takeTree n Leaf = Leaf
takeTree n (Node x l r)  = Node  x (takeTree (n-1) l ) (takeTree (n-1) r )

mapTree :: (a->b) -> Tree a -> Tree b
mapTree f Leaf = Leaf
mapTree f (Node x l r) = Node (f x) (mapTree f l) (mapTree f r)

replicateTree :: Int -> a -> Tree a
replicateTree n = takeTree n . repeatTree
```