Exercises
Declaring Types, Trees

Exercise 1

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 Expression

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

Exercise 2

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

Exercise 3

(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

Exercise 4

Using the following (as seen in class) :

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

together with a function

```
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

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

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

Exercise 5

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
   ```

Exercise 6

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.

Exercise 7

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

New exercise - use trace to help you

```haskell
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
```

Solutions

Solutions to exercise 2

```
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 []
```

Solutions to exercise 3

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

Solutions to exercise 4

```
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.

Solutions to exercise 5

```
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
```

Solutions to exercise 6

```
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
```

Solutions to exercise 7

```
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
```