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

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 ***size*** that returns the number of leaves in a tree.

   ```
   size :: Tree a -> Int
   ```

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

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

Exercise 4

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.