Problems

1. `intersect, intersectTail, and intersectAll`

(a) `intersect` - 6%

The function `intersect` takes two lists, `l1` and `l2`, and returns a list including the elements that exists in both lists. The resulting list should not include any duplicates. The elements in the output can have arbitrary order.
You may use the built in `elem` function or the HW1 `exists` function in your solution.

The type of intersect can be `intersect :: Eq a => [a] -> [a] -> [a]`

Examples:

```
> intersect [2,2,5,6,6,8,9] [1,3,2,2,4,4,5,7,8,10]
[2,5,8]
> intersect [5,6,7,8,9] [8,8,10,10,11,12,5]
[5,8]
> intersect ["a","b","d"] ["c","e","f","g"]
[]
> intersect [1,2,3] []
[]
```

(b) `intersectTail` - 10%

Re-write the `intersect` function from part (a) as a tail-recursive function. Name your function `intersectTail`.
The type of `intersectTail` should also be `Eq a => [a] -> [a] -> [a]`
You may use the same test cases provided above to test your function.

(c) `intersectAll` - 6%

Using `intersect` function defined above and the `foldr (or foldl)` function, define `intersectAll` which takes a list of lists and returns a list containing the intersection of all the sublists of the input list. **Provide an answer using `foldr` (or `foldl`); without using explicit recursion.**
The type of `intersectAll` should be one of the following:

```
intersectAll:: (Foldable t, Ord a) => t [a] -> [a]  OR
intersectAll:: Ord a => [[a]] -> [a]
```

Examples:

```
> intersectAll [[1,3,3,4,5,5,6],[3,4,5],[4,4,5,6],[3,5,6,6,7,8]]
[5]

> intersectAll [[3,4],[-3,-4,3,4],[-3,-4,5,6]]
[ ]

> intersectAll [[3,4,5,5,6],[4,5,6],[],[3,4,5]]
[ ]
```

2. partition - 10%

`partition` function takes a predicate function (`op`) and a list (`iL`) as input, and returns a 2-tuple (`left,right`) as output where `left` is the list of the elements (e_i) in `iL` for which `(op` e_i`)` evaluates to `True`, and `right` is the list of those elements in `iL` for which `(op` e_i`)` evaluates to `False`. The elements of `left` and `right` retain the same relative order they possessed in `iL`.
Your function shouldn't need a recursion but should use the higher order function "`filter`". You may define additional helper function(s), which are not recursive.
The type of the `partition` function should be:

```
partition :: (a -> Bool) -> [a] -> ([a], [a])
```

Examples:

```
> partition (\x -> (x<=4)) [1,7,4,5,3,8,2,3]
([1,4,3,2,3],[7,5,8])
> partition null [[1,2],[1],[],[5],[],[6,7,8]]
([[],[]],[[1,2],[1],[5],[6,7,8]])
> partition (elem 1) [[1,2],[1],[],[5],[],[6,7,8]]
([[1,2],[1]],[[],[5],[],[6,7,8]])
> partition (\x -> (x<=4)) []
([],[])
```

3. sumL, sumMaybe, and sumEither

(a) **sumL - 5%**

Function `sumL` is given a `list` of `lists` and it returns the sum of all numbers in all sublists of the input list. Your function shouldn't need a recursion but should use functions "`map`" and "`foldr`". You may define additional helper functions which are not recursive.
The type of the `sumL` function can be one of the following:

```
sumL :: (Num b) => [[b]] -> b
sumL :: (Num b, Foldable t) => [t b] -> b
```

Examples:

```
> sumL [[1,2,3],[4,5],[6,7,8,9],[]]
45
> sumL [[10,10],[10,10,10],[10]]
60
> sumL [[]]
0
> sumL []
0
```

(b) **sumMaybe - 10%**

Function `sumMaybe` is given a `list` of `Maybe lists` and it returns the sum of all `Maybe` values in all sublists of the input list. Your function shouldn't need a recursion but should use functions "`map`" and "`foldr`". You may define additional helper functions which are not recursive. The type of the `sumMaybe` function can be one of the following:

```
sumMaybe :: (Num a) => [[(Maybe a)]] -> Maybe a
sumMaybe :: (Num a, Foldable t) => [t (Maybe a)] -> Maybe a
```

(Note: To implement `sumMaybe`, change your `sumL` function and your helper function in order to handle `Maybe` values instead of numbers. Assume the integer value for `Nothing` is `0`.)

Examples:

```
> sumMaybe [[(Just 1),(Just 2),(Just 3)],[(Just 4),(Just 5)],[(Just 6),Nothing
],[],[Nothing ]]
Just 21
> sumMaybe [[(Just 10),Nothing],[(Just 10), (Just 10), (Just 10),Nothing,Nothing]]
Just 40
> sumMaybe [[Nothing ]]
Nothing
> sumMaybe []
Nothing
```

(c) **sumEither - 12%**

Define the following Haskell datatype:

```
data IEither  = IString String | IInt Int
                deriving (Show, Read, Eq)
```

Define an Haskell function `sumEither` that takes a list of **`IEither lists`** and it returns an **`IInt`** value which is the sum of all values in all sublists of the input list. The parameter of the **`IString`** values should be converted to integer and included in the sum. You may use the following function to convert a string value to integer.

```
getInt x = read x::Int
```

Your `sumEither` function shouldn't need a recursion but should use functions "`map`" and "`foldr`". You may define additional helper functions which are not recursive. The type of the `sumEither` function can be one of the following:

```
sumEither:: Foldable t => [t IEither] -> IEither
sumEither:: [[IEither]] -> IEither
```

Examples:

```
> sumEither [[IString "1",IInt 2,IInt 3],[IString "4",IInt 5],[IInt 6,IString
"7"],[],[IString "8"]]
IInt 36
> sumEither [[IString "10" , IInt 10],[],[IString "10"],[]]
IInt 30
> sumEither [[]]
IInt 0
```

4. depthScan, depthSearch, addTrees

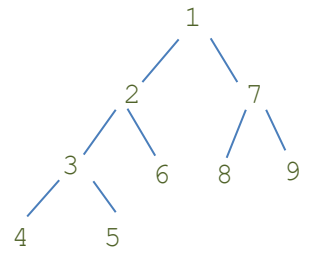
In Haskell, a polymorphic binary tree type with data both at the leaves and interior nodes might be represented as follows:

```
data Tree a = LEAF a | NODE a (Tree a) (Tree a)
              deriving (Show, Read, Eq)
```

(a) depthScan - 10%

Write a function `depthScan` that takes a tree of type `(Tree a)` and returns a list of the `a` values stored in the leaves and the nodes. The order of the elements in the output list should be based on the depth-first order traversal of the tree.

The type of the depthScan function should be: `depthScan :: Tree a -> [a]`



on depth-first order traversal:
`[4,5,3,6,2,8,9,7,1]`

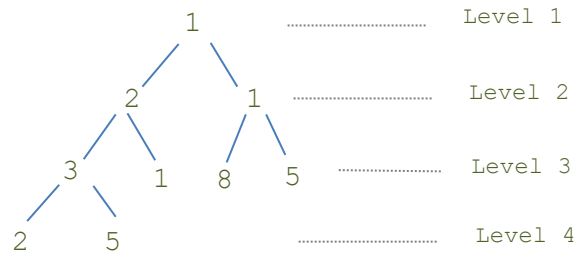
Examples:

```
t1 =  NODE
         "Science"
         (NODE "and" (LEAF "School")(NODE
                                       "Engineering"
                                       (LEAF "of")
                                       (LEAF "Electrical")))
          (LEAF "Computer")
depthScan t1
["School","of","Electrical","Engineering","and","Computer","Science"]

t2 = NODE 1 (NODE 2 (NODE 3 (LEAF 4) (LEAF 5)) (LEAF 6)) (NODE 7 (LEAF 8) (LEAF 9))
depthScan t2
[4,5,3,6,2,8,9,7,1]

depthScan (LEAF 4)
[4]
```

(b) depthSearch – 12%

Write a function `depthSearch` that takes a tree of type (`Tree a`) and a value and returns the level of the tree where the value is found. If the value doesn't exist in the tree, it returns -1. The tree nodes should be visited with depth-first -order traversal and the level of the first matching node should be returned. The type of the depthSearch function can be:

```
depthSearch :: (Ord p, Num p, Eq a) => Tree a -> a -> p
```



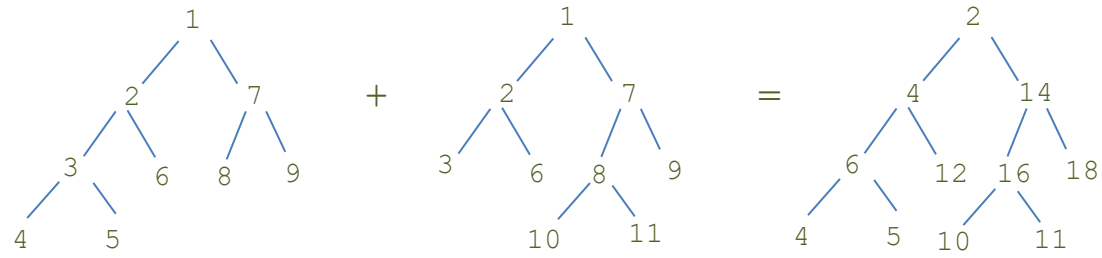

```
t3 = NODE 1 (NODE 2 (NODE 3 (LEAF 2) (LEAF 5)) (LEAF 1)) (NODE 1 (LEAF 8) (LEAF 5))
depthSearch t3 1
3
depthSearch t3 5
4
depthSearch t3 8
3
depthSearch t3 4
-1
```

(c) `addTrees` – 15%

Write a function `addTrees` that takes two `(Tree int)` values and returns an `(Tree int)` where the corresponding nodes from the two trees are added. The trees might have different depth. You should copy particular branches/nodes of the trees if the other tree doesn't have that branch/node. See the example below.

The type of the `addTrees` function can be:

```
addTrees :: Num a => Tree a -> Tree a -> Tree a
```



We can create the above `Tree(s)` as follows:

left :

```
left = NODE 1 (NODE 2 (NODE 3 (LEAF 4) (LEAF 5)) (LEAF 6)) (NODE 7 (LEAF 8) (LEAF 9))
```

right:

```
right = NODE 1 (NODE 2 (LEAF 3) (LEAF 6)) (NODE 7 (NODE 8 (LEAF 10) (LEAF 11)) (LEAF 9))
```

And `addTrees left right` will return the rightmost `(Tree Int)` which is equivalent to the following:

```
NODE 2
(NODE 4 (NODE 6 (LEAF 4) (LEAF 5)) (LEAF 12))
(NODE 14 (NODE 16 (LEAF 10) (LEAF 11)) (LEAF 18))
```

5. **Tree examples – 4%**

Create two trees of type Tree. The height of both trees should be at least 4. Test your functions `depthScan`, `depthSearch`, `addTrees` with those trees. The trees you define should be different than those that are given.

Here is some additional test data.

```
l1 = LEAF "1"
l2 = LEAF "2"
l3 = LEAF "3"
l4 = LEAF "4"
n1 = NODE "5" l1 l2
n2 = NODE "6" n1 l3
t4 = NODE "7" n2 l4
```

```
depthScan t4
```

```
["1","2","5","3","6","4","7"]
```

Testing your functions

We will be using the `HUnit` unit testing package in CptS355. See http://hackage.haskell.org/package/HUnit for additional documentation.

The file `HW2SampleTests.hs` provides at least one sample test case comparing the actual output with the expected (correct) output for each problem. This file imports the `HW2` module (`HW2.hs` file) which will include your implementations of the given problems.

You are expected to add **at least 2 more test cases** for each problem. Make sure that your test inputs cover all boundary cases. Choose test input different than those provided in the assignment prompt.

In `HUnit`, you can define a new test case using the `TestCase` function and the list `TestList` includes the list of all test that will be run in the test suite. So, make sure to add your new test cases to the `TestList` list. All tests in `TestList` will be run through the "`runTestTT tests`" command.

If you don't add new test cases you will be deduced at least 5% in this homework.

Important note about negative integer arguments:

In Haskell, the -x, where x is a number, is a special form and it is a prefix (and unary) operator negating an integer value. When you pass a negative number as argument function, you may need to enclose the negative number in parenthesis to make sure that unary (-) is applied to the integer value before it is passed to the function.

For example: `foo -5 5 [-10,-5,0,5,10]` will give a type error, but

`foo (-5) 5 [-10,-5,0,5,10]` will work