# 1 Guidelines

1. You may do this homework individually or you may do the entire assignment with a homework partner. If you work with another person, you may turn in one set of solutions for both of you.
2. Haskell code is required for some parts in questions 1-6. Code should be submitted electronically according to instructions available in the forum. Solutions for all other questions should be turned in on paper for manual grading.
3. You may use any development environment for questions 1-6. We recommend Leksah that can be obtained from here `http://www.haskell.org/haskellwiki/Leksah` for free. You may also consider EclipseFP , can be obtained from here `http://eclipsefp.github.io/`.

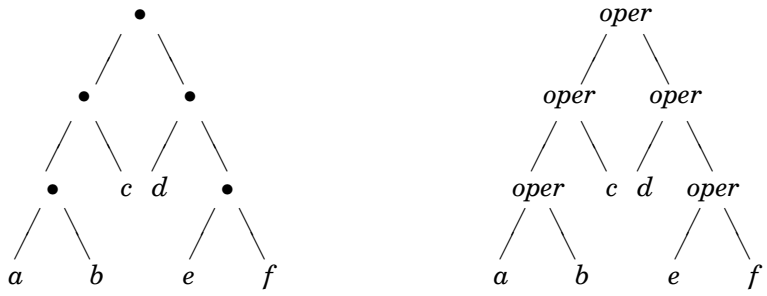
## Problems

### 1. Haskell Reduce for Trees

The binary tree datatype

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

describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node).




- Write a function

  `reduce :: (a → a → a) → Tree a → a`

  that combines all the values of the leaves using the binary operation passed as a parameter. In more detail, if `oper : a → a → a` and `t` is the nonempty tree on the left in this picture, then `reduce oper t` should be the result obtained by evaluating the tree on the right. For example, if `f` is the function

  ```
  f :: Int → Int → Int
  f x y = x + y
  ```

  then `reduce f (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3))` = (1+2)+3 = 6. Explain your definition of `reduce` in one or two sentences. Assume that `oper` is a commutative and associative operator. Also for the case when the tree has just a single Leaf node, the `reduce` function should simply return the contents of this leaf node.
- Write a function `toList :: Tree a → [a]` that returns a list of the elements in the argument tree.
- Write a function `reduceList :: (a → a → a) → [a] → a` that reduces a list according to the supplied function argument.

- Use `toList` and `reduceList` to write a predicate `prop_reduceTest`, check it manually in the interpreter.

A possible layout for the entire code is given below :

```
{-# OPTIONS -XTypeSynonymInstances #-}


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

reduce :: (a->a->a) -> Tree a -> a
reduce f t = < ... >

toList :: Tree a -> [a]
toList t = < ... >

reduceList :: (a->a->a) -> [a] -> a
reduceList f l = < ... >

prop_reduceTest :: (Int->Int->Int) -> TS -> Bool
prop_reduceTest f tree = < ... >
```

You need to fill in the blank spaces (< ... >) that appear in the above code. The above code may not compile on some of the older versions of GHC. For older versions, you could try starting `ghci` with the $-$`fglasgow` $-$ `exts` flag ie running `ghci` $-$ `fglasgow` $-$ `exts`.

## 2. ..... Nonlinear Pattern Matching

Haskell patterns cannot contain repeated variables. This exercise explores this language design decision. A declaration with a single pattern is equivalent to a sequence of declarations using destructors. For example,

```
p = (5,2)
(x,y) = p
```

is equivalent to

```
p = (5,2)
x = fst p
y = snd p
```

where `fst p` is the Haskell expression for the first component of pair `p` and `snd` similarly returns the second component of a pair. The operations `fst` and `snd` are called *destructors* for pairs.

A function declaration with more than one pattern is equivalent to a function declaration that uses standard `if-then-else` and destructors. For example,

```
f [] = 0
f (x:xs) = x
```

is equivalent to

```
f z = if z == [] then 0 else head z
```

where `head` is the Haskell function that returns the first element of a list.

*Questions:*

(a) Write a function that does not use pattern matching and that is equivalent to

```
g (x,0) = x
g (0,y) = y
g (x,y) = x + y
```

Haskell pattern matching is applied in order. When the function `g` is applied to an argument `(a, b)`, the first clause is used if `b = 0`, the second clause if `b` $\neq$ `0` and `a=0`, and the third clause if `b` $\neq$ `0` and `a` $\neq$ `0`.

(b) Consider the following function:

```
eq(x,x) = True
eq(x,y) = False
```

Describe how you could translate Haskell functions that contain patterns with repeated variables, like `eq`, into functions without patterns, using destructors and `if-then-else` expressions. Give the resulting translation of the `eq` function.

(c) Why do you think the designers of Haskell prohibited repeated variables in patterns? (*Hint:* Under what circumstances might the translation you just outlined not work?)

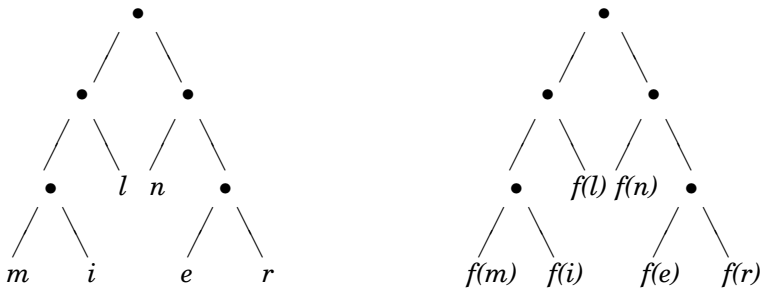
## 3. Haskell Map for Trees

(a) (Submit code) The binary tree data type

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

describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node).

Write a function `maptree` that takes a function as an argument and returns a function that maps trees to trees by mapping the values at the leaves to new values, using the function passed in as a parameter. In more detail, if `f` is a function that can be applied to the leaves of tree `t` and `t` is the tree on the left, then `maptree f t` should result in the tree on the right:




For example, if `f` is the function `f x = x + 1` then

```
maptree f (Node (Node (Leaf 1) (Leaf 2))  (Leaf 3))
```

should evaluate to `Node (Node (Leaf 2) (Leaf 3)) (Leaf 4)`.

(b) (Submit text) Explain your definition in one or two sentences.

(c) (Submit text) What is the type Haskell gives to your function? Why is it not the type `(t -> t) -> Tree t -> Tree t`?

## 4. Currying

This problem asks you to show that the Haskell types `a -> b -> c` and `(a,b) -> c` are essentially equivalent.

(a) (Submit code) Define higher-order Haskell functions

```
curry :: ((a,b) -> c) -> (a -> (b -> c))
```

and

```
uncurry ::  (a -> (b -> c)) -> ((a,b) -> c)
```

(b) (Submit text) For all functions `f ::  (a,b) -> c` and `g ::a -> (b -> c)`, the following two equalities should hold (if you wrote the right functions):

```
uncurry(curry f) = f
curry(uncurry g) = g
```

Explain why each is true for the functions you have written. Your answer can be three or four sentences long. Try to give the main idea in a clear, succinct way. (We are more interested in insight than in number of words.) Be sure to consider termination behavior as well.

## 5. Haskell Type Class Inference

This problem involves the Haskell program listed below:

```
module Main where
import System.Environment
import Data.List

comp l1 l2 = let
  p1 = (length l1, l1)
  p2 = (length l2, l2)
  in compare p1 p2
```

(a) What is the type of `comp`? Explain the inference process that produces this type in English (you do not need to draw an inference diagram). Note that the `compare` function

```
compare :: (Ord a) => a -> a -> Ordering
Ordering =  LT | EQ | GT
```

is like the overloaded function `(<)` except that it returns either LT (for less than) or EQ (for equal) or GT (for greater than) instead of a boolean.

(b) Consider the application

```
-- s1 and s2 are Strings
r1 = comp s1 s2
```

What implementations of `compare` could be involved in computing r1? Explain your answer. You may assume that `compare` for a compound type is implemented recursively in terms of versions for its constituent components.