Tree Equality: The Problem

Recall the tree data type we defined:

```
data LTree a = LLeaf a | LBranch (LTree a) (LTree a)
```

Suppose we want to write a function that determines if two trees are equal:

```
treeEq (LLeaf x) (LLeaf y) = x==y
treeEq (LBranch t1 t2) (LBranch s1 s2) =
  treeEq t1 s1 && treeEq t2 s2
treeEq _ _ = False
```

There are two problems:

1. What is its type?
2. We would like to overload == and not use the name `treeEq`.

Operator Overloading: The Questions

Recall our previous story about numbers: we said

```
3 :: Integer
(+) :: Integer -> Integer -> Integer
```

But this is obviously lying. For example, + also works with `Int`, `Rational`, `Float`, and `Double`. On the other hand, it does not work with lists. So we raise the questions:

- What is the actual type of + then? Is it polymorphic?
- How is this overloading implemented?
- Can we extend this overloading to my data types and my operators?

Type Classes

All of the above problems and questions are resolved in Haskell by *type classes*.

- Conceptually, a type class represents a restricted set of types. (Contrast: a type variable represents the set of all types, unrestricted.)
- Pragmatically, a type class declares a few operators and functions for overloading.

E.g., == is declared in the type class `Eq`:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```

This says: For a type a that belongs to the class `Eq`, it has two operators: == and /= of type `a->a->Bool`.

Type Instances: Declaration

You can put any data type into a type class, but you have to implement the operators.

Put it in another way: in order to overload an operator for your data type, you put it into the appropriate type class.

E.g., recall `Tree`:

```
data Tree = Leaf | Branch Tree Tree
```

Put it into class `Eq`:

```
instance Eq Tree where
  Leaf==Leaf = True
  (Branch t1 t2)==(Branch s1 s2) = t1==s1 && t2==s2
  _==_ = False
```

Type Instances: Defaults

Note that we only implemented == but not /=. This is because Eq has default implementations:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x == y      = not (x/=y)
  x /= y      = not (x==y)
```

If we only implement ==, the above code for /= will work, and vice versa.

So now we can compare trees:

```
Leaf /= Branch Leaf Leaf   --True
Branch Leaf Leaf == Branch Leaf Leaf   --True
```


Types of Overloaded Operators

What is the type of == then? It ain't a->a->Bool, as a could be outside Eq.

Here it is:

```
(==) :: Eq a => a->a->Bool
```

It says: it is of type a->a->Bool assuming that a belongs to Eq.

Likewise, there is a class Num consisting of all numeric types, and we have:

```
(+) :: Num a => a->a->a
3 :: Num a => a
```

So + and 3 will work for Int, Integer, Rational, Float, Double, etc., because they all belong to Num (and they all implement + accordingly).

Tree Equality: Solution

To overload == for LTree:

```
data LTree a = LLeaf a | LBranch (LTree a) (LTree a)
```

we need a prerequisite: a should belong to Eq first.

```
instance Eq a => Eq (LTree a) where
```

This says: LTree a belongs to class Eq provided that a already belongs to class Eq.

```
  (LLeaf x)==(LLeaf y) = x==y
```

That is why we need the Eq a assumption.

```
  (LBranch t1 t2)==(LBranch s1 s2) = t1==s1 && t2==s2
  _==_ = False
```


Tree Equality: Solution 2

Could we still write treeEq and use it? Yes.

```
treeEq (LLeaf x) (LLeaf y) = x==y
treeEq (LBranch t1 t2) (LBranch s1 s2) =
  treeEq t1 s1 && treeEq t2 s2
treeEq _ _ = False
```

The type is

```
Eq a => LTree a -> LTree a -> Bool
```

You can then use it to define == for trees:

```
instance Eq a => Eq (LTree a) where
  (==) = treeEq
```

Tree Equality: Solution 3

Probably 99% of your data types will have == defined in a similar fashion as the above.

Haskell can automatically generate such naïve definitions:

```
data LTree a = LLeaf a | LBranch (LTree a) (LTree a)
  deriving Eq
```

Then you immediately have == and /= defined for you the way above.

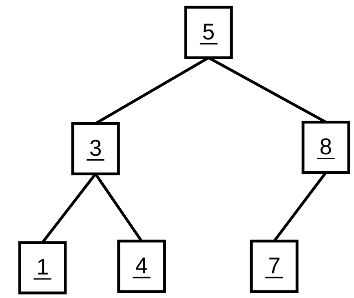
Another class, `Show`, is for types that can be printed. You can derive it and get the naïve printing too:

```
dataLTree a = LLeaf a | LBranch (LTree a) (LTree a)
  deriving (Eq, Show)
```


Binary Search Tree

We can now define polymorphic but restricted data types.

E.g., let us define binary search trees.



The key in a node is greater than all keys in the left subtree, and less than all keys in the right subtree. For simplicity, we disallow duplicate keys.

BST as Constrained Polymorphic Type

We wish to allow any type of keys, but the type must come with the < and the > operators. These come from the `Ord` class (for "ordered"):

```
class Eq a => Ord a where
  (<), (<=), (>=), (>) :: a->a->Bool
  ...
```

It declares the comparison operators. It also requires the type to belong to `Eq` first.

Now we can define our polymorphic binary search tree:

```
data Ord a => BST a = Nil | Node a (BST a) (BST a)
```

We require the key type to come from the `Ord` class. For simplicity, keys go into internal nodes, and "null pointers" are modelled by `Nil`.

BST Operations

The membership operation: does a key occur in the tree?

```
member :: Ord a => a -> BST a -> Bool
member key Nil = False
member key (Node x t s) | key<x = member key t
                        | key>x = member key s
                        | key==x = True
```

The insert operation: add a key to a tree, returning the new tree.

```
insert :: Ord a => a -> BST a -> BST a
insert k Nil = Node k Nil Nil
insert k n@(Node x t s) | k<x = Node x (insert k t) s
                        | k>x = Node x t (insert k s)
                        | otherwise = n
```