CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 09: Basic Type Classes Concluded

- Functor Review
- Creating our own data types and making them part of the Haskell ecosystem: instance declarations
- Class declarations
- Derived instances of classes

Reading: Hutton 8.1 – 8.5

Additional Reading: Build You a Haskell for Great Good, Ch. 8 "Typeclasses 102"

Review: Functors

So far all our type classes have been with basic (non-function) data.

How do we take advantage of the full power of functional programming, that is, how we do make all this higher-order?

Let's examine the `Functor` type class, which provides for map-like functions. Recall that `map` has the type

```
map :: (a -> b) -> [a] -> [b]
```

We would like to provide this kind of functionality for arbitrary data types, not just lists. For example, we'd like to map over `Maybe` or trees or ....

Type Classes: Functors

How to do a version of map which works on Maybe?

For example, over a `Maybe` it would have to be

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

(you'll see why we changed the name in a minute). This would allow us to apply a function inside a `Maybe`:

```
Main> fmap (*2) (Just 5)
Just 10
Main> fmap (*2) Nothing
Nothing
Main> fmap length (Just ”Hi there!“)
Just 9
Main> fmap (++ " Folks!") (Just "Hi There")
Just "Hi There Folks!"
Main> fmap (++ " Folks!")  Nothing
Nothing
```

Type Classes: Functors

This is the purpose of the Functor type class, which is defined as follows:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

This is an example of a type class which doesn't provide any implementation, just requires that any instance must provide an implementation of `fmap`.

What is **`f`** in this declaration? It looks like a function, since it is applied to arguments **`a`** and **`b`**. In fact, it is, but we call it a **type constructor**, since it takes a type (such as `Integer`) and constructs a type based on it:

```
data Maybe a = Nothing | Just a

getPositive :: Integer -> Maybe Integer
getPositive n | n >= 0    = Just n
              | otherwise = Nothing

Maybe Integer  =>  Nothing | Just Integer
```

Type Classes: Functors

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

To create a map on `Maybe` types, we do this:

```
instance Functor Maybe where
    -- fmap :: (a -> b) -> Maybe a -> Maybe b
    fmap g (Just x) = Just (g x)
    fmap g Nothing  = Nothing
```

Notice carefully that we did not say

```
instance Functor (Maybe a) where
```

Functor wants a type constructor, not a type! By referential transparency, you can't substitute `(Maybe a)` for `f`, but you can substitute `Maybe`:

~~`(Maybe a) a`~~ ~~`(Maybe a) b`~~ `Maybe a` `Maybe b`

Type Classes: Functors

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b


instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing

Main> fmap (*8) [2,3,4,4]
[16,24,32,32]

Main> fmap (*8) (Just 3)
Just 24
```

Why do this? Now we can use the fmap function, and also, any other function which uses fmap can deal with our data type (we will see how this works with Monads).

Your data type has joined the Haskell "ecosystem" and can use all of its many features!

Type Classes: Functors

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

We can create an `fmap` for any data type we create, by simply providing the appropriate implementation of `fmap` when we make our data type an instance of the `Functor` type class:

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

instance Functor Tree where
    -- fmap :: (a -> b) -> Tree a -> Tree b
    fmap g Null = Null
    fmap g (Node left x right)
       = Node (fmap g left) (g x) (fmap g right)

Main> fmap (*2) Null
Null
Main> (foldr insert Null [5,7,3,12])
Node (Node Null 3 Null) 5 (Node Null 7 (Node Null 12 Null))

Main> fmap (*2) (foldr treeInsert Null [5,7,3,2,1,7])
Node (Node Null 6 Null) 10 (Node Null 14 (Node Null 24 Null))
```

Type Classes: Creating Types

Reading: Hutton Ch. 8.5

But you can make your data type an instance of **any type class**, not just `Functor`, as long as you use an instance declaration which provides implementations for all the class's functions:

For example, if we define a `Tree` as before, we can't apply == to its instances:

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

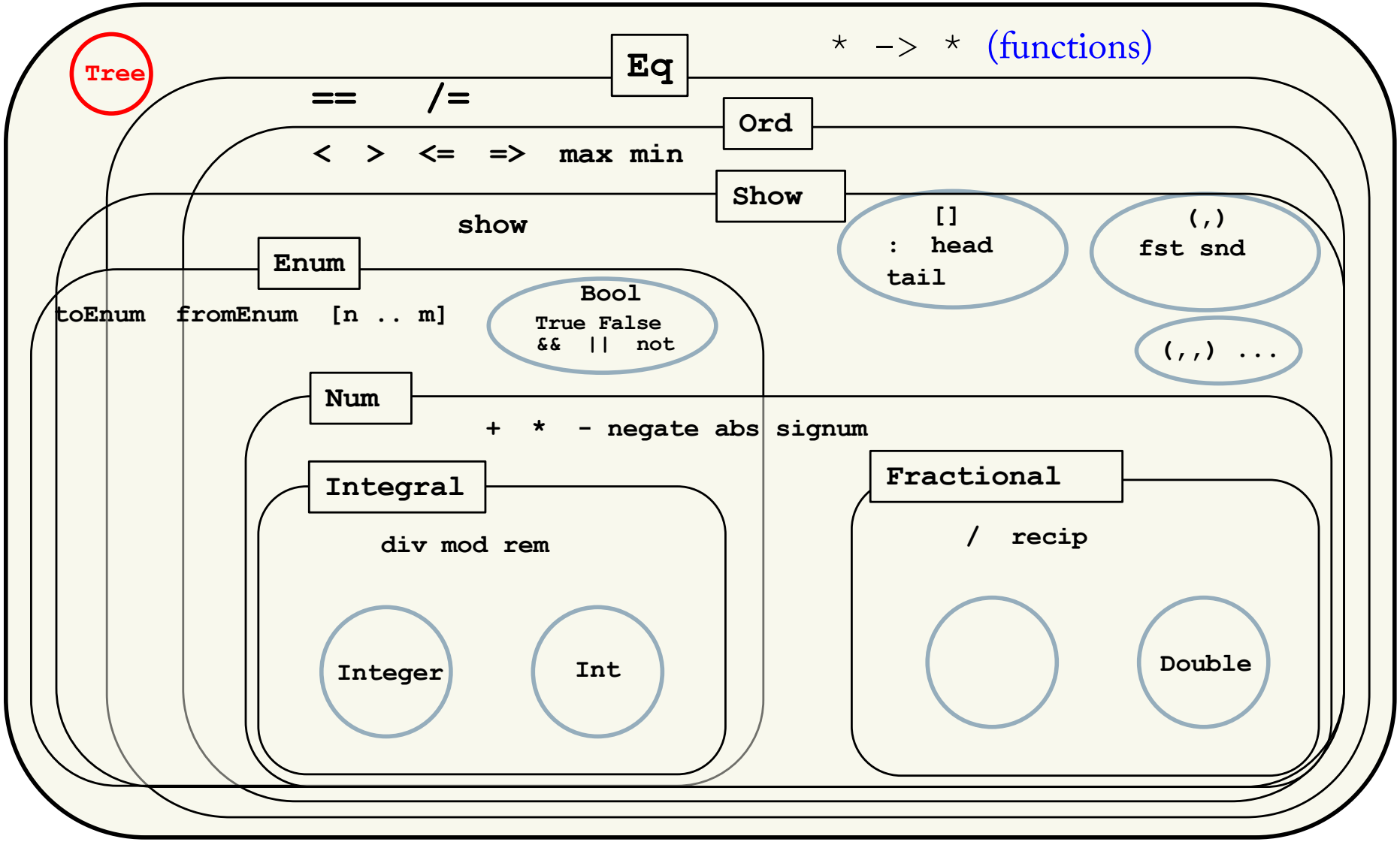
```
Main> Null == Null

<interactive>:16:1: error:
    • No instance for (Eq (Tree a0)) arising from a use of '=='
    • In the expression: Null == Null
      In an equation for 'it': it = Null == Null
```

In fact, we can't even look at the data instances because it is not an instance of Show!

```
Main> Null

<interactive>:13:1: error:
    • No instance for (Show (Tree a0)) arising from a use of 'print'
    • In a stmt of an interactive GHCi command: print it
```

Type Classes: Creating Types

This is because Tree is not an instance of the standard classes -- it can use its own functions, but it can't use their functions! It's a Bare Bones Tree!



Instance Declarations

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

So we declare `Tree` to be an instance of `Eq` and define == on it:

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

instance Eq a => Eq (Tree a) where
   Null == Null = True
   (Node left x right) == (Node left2 x2 right2) =
      (x == x2) && (left == left2) && (right == right2)
   _ == _  = False
```

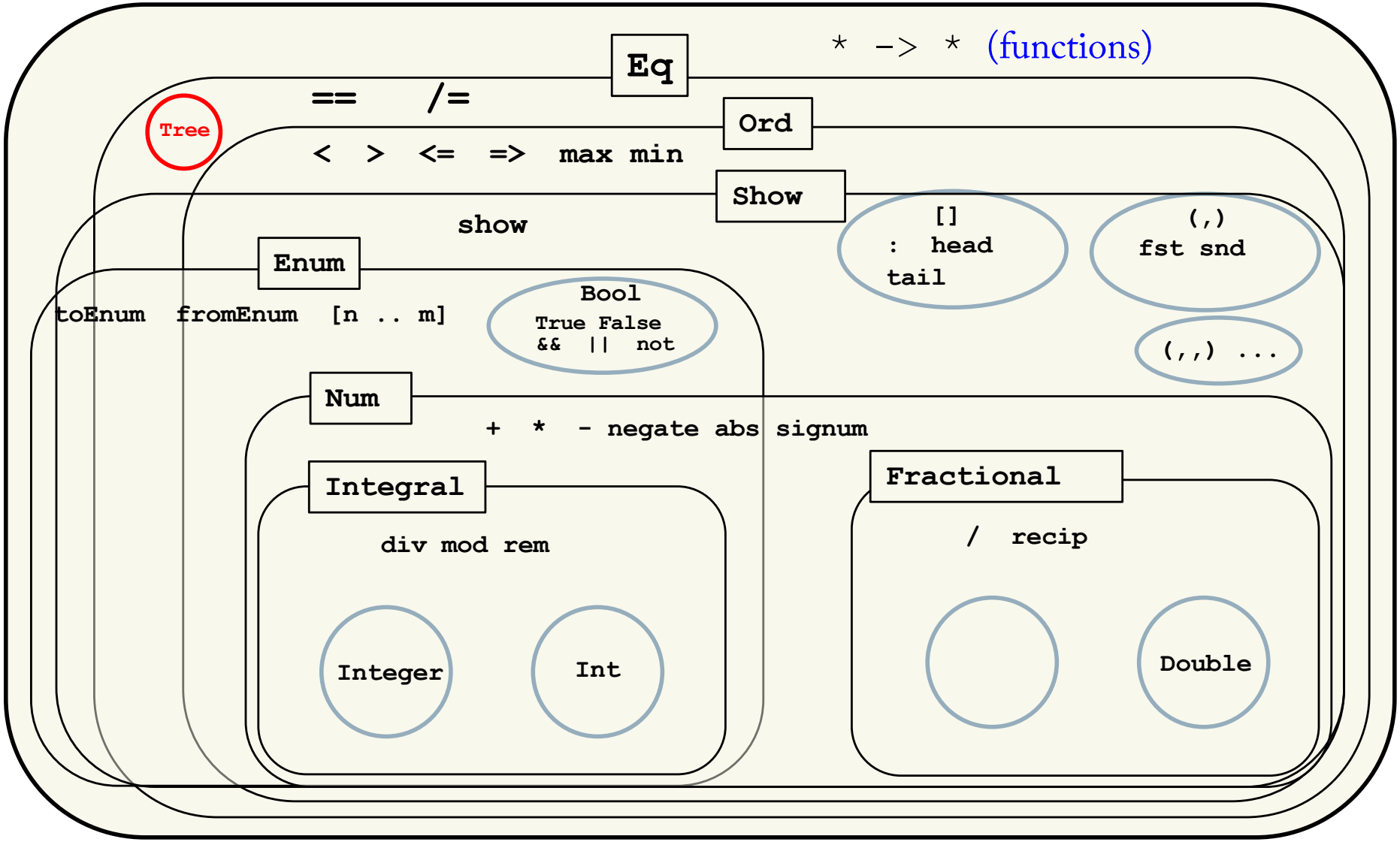
```
Main> Null == Null
True

Main> (Node Null 3 Null) == (Node Null 3 Null)
True

Main> (Node Null 3 Null) == (Node Null 5 Null)
False

Main> Null <= Null
```

But still can't use functions in other classes such as Ord.

```
<interactive>:24:1: error:
    • No instance for (Ord (Tree a0)) arising from a use of '<='
    .....
```

Type Classes: Creating Types

Now Tree is a member of the `Eq` class and can be manipulated by any functions that use == and /= :



Instance Declarations

```haskell
data Tree a = Null | Node (Tree a) a (Tree a)

instance Eq a => Eq (Tree a) where
   Null == Null = True
   (Node left x right) == (Node left2 x2 right2) =
      (x == x2) && (left == left2) && (right == right2)
   _ == _  = False
```

Also, **now you get other functions....** note that `Eq` defined `/=` from `==` so `Tree` can inherit the function `/=` from `Eq`:

```
Main> (Node Null 3 Null) /= (Node Null 5 Null)
True
```

And, we can use any function that only depends on == :

```
Main> Null `elem` [ (Node Null 3 Null),  Null ]
True
```

```
Main> filter (/= Null) [(Node Null 3 Null),Null,(Node Null 2 Null)]
[( () 3 () ),( () 2 () )]
```

This is the whole point of making your data types instances of standard classes: **You can make your data type part of the whole Haskell ecosystem and use all its features.**

Otherwise, you're basically stuck with Bare Bones Haskell!

Instance Declarations

Reading: Hutton Ch. 8.5

Now let's fix the problem that we can't look at the trees, by making `Tree` an instance of the `Show` class; since show just turns trees into Strings, we can choose any way we want of printing out the trees:

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

instance Show a => Show (Tree a) where
  -- show :: Tree a -> string
  show Null = "()"
  show (Node left x right) =
     "( " ++ show left ++ " " ++ show x ++ " " ++ show right ++ " )"
```

Note that there is a bit of recursion going on here: the base type of the Tree must also be an instance of Show.

Call to show on type **a**.

Recursive call to this show.

```
Main> Null
()

Main> (Node Null 2 Null)
( () 2 () )

Main> Node (Node Null 2 Null) 5 (Node Null 9 Null)
( ( () 2 () ) 5 ( () 9 () ) )
```

Class Declarations

This material is taken directly from Hutton Ch. 8.5

We've seen that we can make a data type an instance of an existing class, so that we can use its functions and participate in its part of the Haskell ecosystem. But can we make our own classes? Well, of course.....

A **new type class** can be declared using Haskell's **`class`** declaration; in fact, if you check out the Prelude (Hutton, Appendix B), you will see declarations of the standard classes discussed last time, starting with:

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

Note: The class declaration specifies:

- The names and types of all the functions which define the class; and
- The implementation of some number (perhaps all perhaps none) of these functions.

Instances will

- Inherit the function types and implementations;
- Must provide implementations for any functions not implemented in the class def; and
- May override any implementations – But may not override the types!

Class Declarations

Reading: Hutton Ch. 8.5

Classes can also be extended. For example, `Ord` is declared in the Prelude to extend `Eq`:

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

   min x y | x <= y     = x
           | otherwise  = y
   max x y | x <= y     = y
           | otherwise  = x
```

For a type to be an instance of `Ord` it must be also be an instance of `Eq` and also give implementations of the 6 operators shown above; but since default definitions for 2 of them are already given , you only need to give the missing 4:

```
instance Ord Bool where
   False < True = True
   _     < _    = False

   b <= c = (b < c) || (b == c)
   b > c  = c < b
   b >= c = c <= b
```

Derived Instances

Reading: Hutton Ch. 8.5

Is there any way to avoid all this work when you create a data type? Can't Haskell help out?

Well, yes, the **deriving** mechanism allows you to do this in a simple way as long as you are willing to live with the default implementations that Haskell provides.

```
data Tree a = Null | Node (Tree a) a (Tree a) deriving (Eq,Show)
```

```
Main> Null
Null

Main> (Node Null 2 Null)
Node Null 2 Null

Main> Null == Null
True

Main> (Node Null 4 Null) == (Node Null 4 Null)
True

Main> Null `elem` [ (Node Null 3 Null), Null ]
True

Main> filter (/= Null)  [ (Node Null 3 Null), Null, (Node Null 2 Null) ]
[Node Null 3 Null,Node Null 2 Null]
```

Compare with our home-brewed version of show:

```
Main> (Node Null 2 Null)
( () 2 () )
```

Derived Instances

Reading: Hutton Ch. 8.5

This is pretty standard for ordinary data types, because the default implementations are fine; so the `Bool` data type is actually defined like this:

```
data Bool = False | True  deriving (Eq, Ord, Show, Read)
```

Note: When you use `deriving`, any component types used in your data declaration must already have these types:

```
data Tree a = Null | Node (Tree a) a (Tree a) deriving (Eq,Show)

data Weekday = M | T | W | R | F
```

`Weekday` is not an instance of `Eq` or `Show`!

```
*Main> Null
Null
*Main> (Node Null M Null)

<interactive>:57:1: error:
    • No instance for (Show Weekday) arising from a use of ‘print’
        .....
*Main> Null == Null
True
*Main> (Node Null M Null) == (Node Null M Null)

<interactive>:59:1: error:
    • No instance for (Eq Weekday) arising from a use of ‘==’
```

Derived Instances

Reading: Hutton Ch. 8.5

But this is, of course, easy to fix if you're fine with the default implementations:

```
data Tree a = Null | Node (Tree a) a (Tree a) deriving (Eq,Show)

data Weekday = M | T | W | R | F deriving (Eq,Show)
```

```
Main> (Node Null M Null)
Node Null M Null
Main> (Node Null M Null) == (Node Null M Null)
True
```

Or you may want to implement only some of the classes:

```
data Tree a = Null | Node (Tree a) a (Tree a) deriving (Eq,Show)

data Weekday = M | T | W | R | F deriving Eq

instance Show Weekday where
  show M = "Monday"
  show T = "Tuesday"
  show W = "Wednesday"
  show R = "Thursday"
  show F = "Friday"
```

```
Main> (Node Null M Null)
Node Null Monday Null
```

Just remember the rule:

If you make a parameterized data type an instance of a class, then all the types used by the data type must be instances of the class. Recursion!

Derived Instances

Reading: Hutton Ch. 8.5

The class `Read` is an interesting case, because the default implementation just expects the same syntax as the default `Show` would display:

```
data Tree a = Null | Node (Tree a) a (Tree a) deriving (Eq,Show,Read)

data Weekday = M | T | W | R | F deriving (Eq,Read,Show)
```

```
Main> read "Null" :: Tree Integer
Null

Main> read "Node Null 4 Null" :: Tree Integer
Node Null 4 Null

Main> read "M"
*** Exception: Prelude.read: no parse

Main> read "M" :: Weekday
M
```

Even though `Weekday` is not a polymorphic data type, `read` still can not figure out the type, so it must be supplied!

Derived Instances

Reading: Hutton Ch. 8.5

But of course I can still define my own Show for Weekday:

```
data Tree a = Null | Node (Tree a) a (Tree a) deriving (Eq,Show,Read)

data Weekday = M | T | W | R | F deriving (Eq,Read)


instance Show Weekday where
  show M = "Monday"
  show T = "Tuesday"
  show W = "Wednesday"
  show R = "Thursday"
  show F = "Friday"
```

```
Main> read "M" :: Weekday
Monday
```

Parsing (Lab 04)

Reading: Hutton Ch. 8.5

Digression: What if we want to define a `Read` that reads our own `Tree` representation, perhaps the same as our own `Show` implementation?

This is difficult, because reading a `String` character by character to reconstruct a type (possibly recursive) is not a simple process. For example, it needs to match parentheses in our `String` representation of `Trees` from before:

```
( ( () 2 () ) 5 ( () 9 () ) )
```

This is called parsing, which I hear you know something about after today's lab. We'll be spending more time on parsing when we start to implement languages after Spring Break....