

# Intro to Haskell Notes: Part 11

Adrian Brasoveanu\*

October 14, 2013

## Contents

<b>1</b>	<b>Derived instances</b>	<b>1</b>
1.1	Deriving <i>Eq</i>	2
1.2	Deriving <i>Show</i> and <i>Read</i>	3
1.3	Deriving <i>Ord</i>	5
1.4	Deriving <i>Enum</i> and <i>Bounded</i>	6
<b>2</b>	<b>Type synonyms</b>	<b>8</b>
<b>3</b>	<b>Recursive data types</b>	<b>8</b>
3.1	Lists as recursive data types	8
3.2	Manually deriving instances of <i>Show</i>	10
3.3	Defining infix functions for our data types	11
<b>4</b>	<b>A more complex recursive data type: Binary search trees</b>	<b>12</b>
4.1	Defining binary search trees and manually deriving <i>Show</i> for them	13
4.2	Inserting elements in trees	13
4.3	Building a tree from a list	14
4.4	Searching for elements in trees	15

## 1 Derived instances

We already explained the basics of typeclasses:

- a typeclass is like an interface that defines a particular kind of behavior
- a type can be made an instance of a typeclass if it supports that behavior

For example, the *Int* type is an instance of the *Eq* typeclass because integers can be equated. In particular, this means that *Int* values can be used with the equality  $\equiv$  and non-equality  $\neq$  functions or with any other functions that make use of (non-)equality internally.

Haskell typeclasses should not be confused with classes in a language like Python, for example. In Python, classes are a blueprint from which we can create objects of particular kinds that can do some actions, that allow certain actions to be done to them, that can store variables internally and / or modify external variables etc. Python classes are more like Haskell types.

In contrast, Haskell typeclasses are like interfaces (so maybe more like Python metaclasses): we don't make data from typeclasses, instead we first make our data type and then we think about what it

---

\*Based primarily on *Learn You a Haskell for Great Good!* by Miran Lipovača, <http://learnyouahaskell.com/>.

can act like. If it can act like something that can be equated, we make it an instance of the *Eq* typeclass. If it can act like something that can be ordered, we make it an instance of the *Ord* typeclass etc.

We will see in a latter section how we can manually make our types instances of typeclasses by implementing the ‘interface’ functions associated with the typeclasses.

But for now we’ll just focus on how Haskell can automatically make our type an instance of any of the following typeclasses: *Eq*, *Ord*, *Enum*, *Bounded*, *Show* and *Read*.

Haskell automatically derives the behavior associated with these typeclasses if we use the **deriving** keyword when making our data type. For example, consider this data type:

```
ghci 1> data Person = Person {firstName :: String, lastName :: String, age :: Int }
```

This data type describes persons. Let’s assume that no two people have the same combination of first name, last name and age in our intended application, hence this is an appropriate way to model ‘person’ values.

## 1.1 Deriving *Eq*

If we have records for two people, i.e., we have two values of type *Person*, it makes sense to ask if they represent the same person.

We can try to equate them and see if they’re equal or not, so it would make sense for this type to be part of the *Eq* typeclass.

This is how we derive that our *Person* type is an instance of the *Eq* typeclasses, i.e., that it supports (non-)equality behavior:

```
ghci 2> data Person = Person {firstName :: String, lastName :: String, age :: Int } deriving (Eq)
```

When we derive the *Eq* instance for a type and then try to compare two values of that type with the  $\equiv$  or  $\neq$  functions, Haskell will check:

- first, if the value constructors match; there’s only one value constructor here, so this is not an issue;
- second, if all the data inside the value constructors can be matched; in particular, the values have to be identical for each field *firstName*, *lastName* and *age*.

Thus, to derive an *Eq* instance for a type, the types of all the fields also have to be part of the *Eq* typeclass.

This is not a problem for our *Person* type: both *String* and *Int* are members of the *Eq* typeclass. Let’s test our *Eq* instance:

```
ghci 3> let mikeD = Person {firstName = "Michael", lastName = "Diamond", age = 43 }
```

```
ghci 4> let adRock = Person {firstName = "Adam", lastName = "Horovitz", age = 41 }
```

```
ghci 5> let mca = Person {firstName = "Adam", lastName = "Yauch", age = 44}
```

```
ghci 6> mca == adRock
False
```

```
ghci 7> mikeD == adRock
False
```

```
ghci 8> mikeD == mikeD
True
```

```
ghci 9> mikeD == Person {firstName = "Michael", lastName = "Diamond", age = 43}
True
```

Since *Person* is now in *Eq*, we can automatically use it as the *a* for all functions that have an *Eq a* class constraint in their type signature. Consider, for example, the *elem* function:

```
ghci 10> let beastieBoys = [mca, adRock, mikeD]
```

```
ghci 11> mikeD ∈ beastieBoys
True
```

## 1.2 Deriving *Show* and *Read*

The *Show* and *Read* typeclasses are for things that can be converted to and from strings, respectively.

Just as for *Eq* above, if the value constructors of our type have fields, the types of the fields have to also be part of the *Show* and *Read* typeclasses if we want our type to be a member of these typeclasses.

So let's make our *Person* type a member of *Show* and *Read* as well:

```
ghci 12> data Person = Person {firstName :: String, lastName :: String, age :: Int}
        deriving (Eq, Show, Read)
```

Now we can display a person in ghci:

```
ghci 13> let mikeD = Person {firstName = "Michael", lastName = "Diamond", age = 43}
```

```
ghci 14> mikeD
Person {firstName = "Michael",lastName = "Diamond",age = 43}
```

```
ghci 15> Person {firstName = "Elmo",lastName = "NA",age = 0}
Person {firstName = "Elmo",lastName = "NA",age = 0}
```

```
ghci 16> "mikeD is: " ++ show mikeD
"mikeD is: Person {firstName = \"Michael\", lastName = \"Diamond\", age = 43}"
```

Had we tried to display a person in ghci before deriving a *Show* instance for our *Person* type, Haskell would have complained saying it doesn't know how to represent a person as a string.

*Read* is pretty much the inverse typeclass of *Show*. *Show* is for converting values of our type to a string, *Read* is for converting strings to values of our type.

When we use the *read* function though, we have to use an explicit type annotation to tell Haskell which type we want to get as a result. If we don't make the type explicit, Haskell won't know which type we want.

```
ghci 17> read "Person {firstName =\"Elmo\", lastName =\"NA\", age = 0}" :: Person
Person {firstName = "Elmo",lastName = "NA",age = 0}
```

Compare with the call without a specified type:

```
ghci 18> read "Person {firstName =\"Elmo\", lastName =\"NA\", age = 0}"
```

If we use the result of our *read* later on in a way that Haskell can infer that it should read it as a person, we don't have to use type annotation:

```
ghci 19> read "Person {firstName =\"Elmo\", lastName =\"NA\", age = 0}" == mikeD
False
```

We can also read parameterized types, but we have to fill in the type parameters. For example, we can't do:

```
ghci 20> read "Just 'f'" :: Maybe a
```

But we can do:

```
ghci 21> read "Just 'f'" :: Maybe Char
Just 'f'
```

### 1.3 Deriving *Ord*

We can also derive instances for the *Ord* typeclass, which is the class of types whose values can be ordered.

If we compare two values of the same type that were made using different value constructors, the value which was made with the constructor that's defined first is considered smaller.

For example, consider the following definition of the *Bool* type:

(1) **data** *Bool* = *False* | *True* **deriving** (*Ord*)

Because the *False* value constructor is specified first and the *True* value constructor is specified after it, *True* is greater than *False*:

```
ghci 22> True 'compare' False
GT
```

```
ghci 23> True < False
False
```

```
ghci 24> True > False
True
```

The same reasoning applies to the definition of the *Maybe a* type:

- the *Nothing* value constructor is specified before the *Just* value constructor, so a value of *Nothing* is always smaller than a value of *Just* something;
- but if we compare two *Just* values, then the result depends on the values inside of them.

```
ghci 25> Nothing < Just 100
True
```

```
ghci 26> Nothing > Just (-49999)
False
```

```
ghci 27> Just 50 < Just 100
True
```

```
ghci 28> Just 50 > Just 100
False
```

```
ghci 29> Just 3 'compare' Just 2
GT
```

But we can't do something like *Just (\*3) < Just (\*2)*, because *(\*3)* and *(\*2)* are functions, which aren't instances of *Ord*.

```
ghci 30> Just (*3) < Just (*2)
```

## 1.4 Deriving *Enum* and *Bounded*

We can easily use algebraic data types to make enumerations and the *Enum* and *Bounded* typeclasses help us with that.

Consider the following data type:

```
ghci 31> data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

Because all the value constructors are nullary / have an arity of 0 / take no parameters / have no fields, we can easily make *Day* part of the *Enum* typeclass. Recall that the *Enum* typeclass is for things that have predecessors and successors.

We can also make it part of the *Bounded* typeclass, which is for things that have a lowest possible value and a highest possible value.

And while we're at it, let's also make it an instance of all the other typeclasses we discussed up until now and see what we can do with it.

```
ghci 32> data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
         deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

Because it's part of the *Show* and *Read* typeclasses, we can convert values of this type to and from strings:

```
ghci 33> Wednesday
Wednesday
```

```
ghci 34> show Wednesday
"Wednesday"
```

```
ghci 35> read "Saturday" :: Day
Saturday
```

Because it's part of the *Eq* and *Ord* typeclasses, we can equate or compare days:

```
ghci 36> Saturday == Sunday
False
```

```
ghci 37> Saturday == Saturday
True
```

```
ghci 38> Saturday > Friday
True
```

```
ghci 39> Saturday < Friday
False
```

```
ghci 40> Monday `compare` Wednesday
LT
```

*Day* is also a member of *Bounded*, so we can get the lowest and highest day:

```
ghci 41> minBound :: Day
Monday
```

```
ghci 42> maxBound :: Day
Sunday
```

And it's also an instance of *Enum*, so we can get predecessors and successors of days and we can make list ranges:

```
ghci 43> succ Monday
Tuesday
```

```
ghci 44> pred Saturday
Friday
```

```
ghci 45> [Thursday..Sunday]
[Thursday, Friday, Saturday, Sunday]
```

```
ghci 46> [minBound..maxBound] :: [Day]
[Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday]
```

## 2 Type synonyms

We mentioned previously that the `[Char]` and `String` types are equivalent and interchangeable. That's implemented with type synonyms.

Type synonyms don't really do anything *per se*, they're just a way of giving different names to certain types so that they make more sense to someone reading our code and documentation.

Here's how the standard library defines `String` as a synonym for `[Char]`:

```
(2) type String = [Char]
```

Thus, type synonyms are introduced with the **type** keyword. The keyword might be misleading because we're not actually making a new type (we do that with the **data** keyword), we're simply making an alias for an already existing type.

If we make a function that converts a string to uppercase and call it `toUpperString`, for example, we can give it a type declaration of `toUpperString :: [Char] → [Char]` or `toUpperString :: String → String`. Both of these are essentially the same, only the latter is nicer to read.

Type synonyms can also be parameterized. For example, if we want a type that represents an association list but still want it to be general so that we can use any type as the keys and values, we can do this:

```
ghci 47> type AssocList k v = [(k,v)]
```

*AssocList* is a type constructor that takes two types and produces a concrete type, e.g., `AssocList Int String`.

Recall that concrete types are fully applied types and values always belong to a concrete type.

Now that we have our parametrized type synonym, we can provide a very general but still readable type signature for a lookup function that gets a value by looking up a key in an association list:  $(Eq\ k) \Rightarrow k \rightarrow AssocList\ k\ v \rightarrow Maybe\ v$ . Optional hw question: why do we use *Maybe v* instead of simply *v* as our return type?

## 3 Recursive data types

We've seen that a value constructor in an algebraic data type can have several fields (or none at all), and each field must be of some concrete type.

With that in mind, we can make types whose constructors have fields that are of the same type. Using that, we can create recursive data types, i.e., recursive data structures, where one value of a type contains more values of that same type, which in turn contain more values of the same type and so on.

### 3.1 Lists as recursive data types

Think about this list: `[5]`. That's just syntactic sugar for `5 : []`:

- there's a value on the left side of the `:` operator, namely 5;
- but on the right side, we have another list; in this case, it's an empty list.

Now consider the list `[4,5]`. That desugars to `4 : (5 : [])`. Looking at the first `:`, we see that:

- it also has an element on its left side, namely 4;
- and it has a list `(5 : [])` on its right side.

And the same goes for a list like `3 : (4 : (5 : (6 : [])))`, which could be written either like that, or as `3 : 4 : 5 : 6 : []` (because `:` is right-associative), or even more simply as `[3,4,5,6]`.

Based on the above example, we can provide a general, recursive characterization of what a list is:



- a list can be an empty list, or
- it can be an element prepended (by means of the constructor `:`) to another list; this list can be empty or not.

Let's use algebraic data types to implement our own list type:

```
ghci 48> data List a = Empty | Cons a (List a)
           deriving (Show, Read, Eq, Ord)
```

This reads just like our recursive definition of lists above: it's either an empty list or a combination of a head value and a list. We combine a value and list to form another list by means of the value constructor `Cons`, of arity 2.

If you're confused about this data type definition, you might find it easier to understand if we use record syntax.

```
ghci 49> data List a = Empty | Cons { listHead :: a, listTail :: List a }
           deriving (Show, Read, Eq, Ord)
```

The `Cons` constructor has two fields: one field is of type `a` and the other is of type `List a` or, using the usual Haskell notation, `[a]`.

Remember that `Cons` is just another word for the Haskell prepending operator `:`, which is really a constructor that takes a value and another list and returns a list.

We can already use our new list type. So let's see some examples using the second definition, i.e., the one with record syntax and therefore with a more explicit annotation.

```
ghci 50> Empty
Empty
```

We will call our `Cons` constructor in an infix manner to emphasize its similarity to the prepending operator `:`.

```
ghci 51> 5 `Cons` Empty
Cons { listHead = 5, listTail = Empty }
```

```
ghci 52> 4 `Cons` (5 `Cons` Empty)
Cons { listHead = 4, listTail = Cons { listHead = 5, listTail = Empty } }
```

```
ghci 53> 3 `Cons` (4 `Cons` (5 `Cons` Empty))
Cons { listHead = 3, listTail = Cons { listHead = 4, listTail = Cons { listHead = 5, listTail = Empty } } }
```

Now let's see the same examples with the first definition. We first switch back to it:

```
ghci 54> data List a = Empty | Cons a (List a)
           deriving (Show, Read, Eq, Ord)
```

```
ghci 55> Empty
Empty
```

```
ghci 56> 5 'Cons' Empty
Cons 5 Empty
```

```
ghci 57> 4 'Cons' (5 'Cons' Empty)
Cons 4 (Cons 5 Empty)
```

```
ghci 58> 3 'Cons' (4 'Cons' (5 'Cons' Empty))
Cons 3 (Cons 4 (Cons 5 Empty))
```

We will make extensive use of recursive data types to implement the logical systems we need for natural language semantics.

### 3.2 Manually deriving instances of *Show*

We called our *Cons* constructor in an infix manner so that we can more easily see that is exactly like the prepending operator `:`. And since *Empty* is like `[]`, we can easily see the parallel between more complex examples, e.g., `4 'Cons' (5 'Cons' Empty)`, and their standard Haskell representations, e.g., `4 : (5 : [])`.

But when our custom lists are displayed in `ghci`, *Cons* is still displayed as a prefix operator. If we want it displayed as an infix, we can manually derive an instance of *Show* for our *List a* type.

To do that, we first declare our data type without deriving an instance of *Show* for it:

```
ghci 59> data List a = Empty | Cons a (List a)
           deriving (Read, Eq, Ord)
```

We will now manually derive an instance of *Show*.

We manually derive an instance of a typeclass by defining the ‘interface’ functions associated with that typeclass.

In the case of *Show*, there is only one function: *show*. The function definition is provided after the **instance** keyword, as shown below:

```
ghci 60> instance (Show a) => Show (List a) where {
    show Empty = "#";
    show (Cons x xs) = show x ++ ", " ++ show xs }
```

The curly braces after **where** surrounding the definition block, as well as the semicolon separating the different clauses of the definition are not needed when we're not working in `ghci` directly. Only proper block alignment is required when we manually derive instances in a script / module:

```
(3) instance (Show a) => Show (List a) where
    show Empty = "#"
    show (Cons x xs) = show x ++ ", " ++ show xs
```

Note that we require the type *a* of list elements to also be an instance of the *Show* type class. This is because the function *show* has to be able to take list elements as arguments, as shown in the *show (Cons x xs)* clause above.

Here are the same examples as above, only this time displayed in the particular manner we specified:

```
ghci 61> Empty
#
```

```
ghci 62> 5 'Cons' Empty
5, #
```

```
ghci 63> 4 'Cons' (5 'Cons' Empty)
4, 5, #
```

```
ghci 64> 3 'Cons' (4 'Cons' (5 'Cons' Empty))
3, 4, 5, #
```

We will make extensive use of such *Show* instances in many of our implementations of natural language semantics systems: we will want the formulas, trees, derivations etc. to be displayed in a familiar format.

### 3.3 Defining infix functions for our data types

We can define functions to automatically be infix operators by assigning them names consisting only of special characters.

For example, let's make a function that adds two lists together. This is how `++` is defined for normal lists:

```
(4) infixr 5 ++
    (++) :: [a] -> [a] -> [a]
    [] ++ ys = ys
    (x : xs) ++ ys = x : (xs ++ ys)
```

So we'll just steal that for our own list. We'll name the function `.++` so that it is an infix by default:

```
ghci 65> let { infixr 5 .++ ;
    ( .++ ) :: [a] -> [a] -> [a];
    [] .++ ys = ys;
    (x : xs) .++ ys = x : (xs .++ ys) }
```

We notice a new syntactic construct, the fixity declarations. When we define functions as operators, we can use that to set their precedence level and their associativity behavior (but we don't have to).

Thus, a fixity declaration states how tightly the operator binds and whether it's left-associative or right-associative.

For instance, `*`'s fixity is **infixl** 7 and `+`'s fixity is **infixl** 6:

```
ghci 66> :i (*)
class Num a where ... (*) :: a -> a -> a ... -- Defined in 'GHC.Num' infixl 7 *
```

```
ghci 67> :i (+)
class Num a where (+) :: a -> a -> a ... -- Defined in 'GHC.Num' infixl 6 +
```

This means that both operators are left-associative, e.g.,  $4 * 3 * 2$  is equivalent to  $(4 * 3) * 2$ , and that `*` binds tighter than `+` because it has a higher precedence level, e.g.,  $5 + 4 * 3$  is equivalent to  $5 + (4 * 3)$ . Our concatenation operator `.++` works as expected:

```
ghci 68> "hello, " .++ "new function!"
"hello, new function!"
```

## 4 A more complex recursive data type: Binary search trees

We are now going to implement binary search trees:

- these trees are fundamentally the same kind of structures we need to define formulas in the various formal semantics systems we will study, and also to define syntactic structures for English expressions;
- but the terminal nodes in these binary search trees will be much simpler (we'll just have numbers), so that we can focus on the hierarchical tree structure itself.

Binary search trees are binary trees whose elements are ordered in a particular way to facilitate search:

- they are structures whose non-terminal elements / nodes point to two elements, one on the left and one on the right;
- and the element on the left, i.e., the left daughter node, is always smaller than mother node (relative to some background ordering), while the right daughter node is always bigger than the mother node (relative to the same background ordering);
- each of the daughter nodes can also point to two other nodes or simply to the 'empty tree'; in effect, each node has up to two sub-trees.

Binary search trees are used to implement various data structures to make them more efficient to access and modify.

For example, the sets and maps made available by the *Data.Set* and *Data.Map* modules are implemented using trees, only instead of regular binary search trees, they use *balanced* binary search trees, which are always balanced: their left and right sides are always (about) the same size, which makes them very efficient to manipulate.

But we'll just implement binary search trees and won't worry about balancing them.

The important thing about binary search trees is that all the elements in the left sub-tree of, say, 5 are going to be smaller than 5, while elements in its right sub-tree are going to be bigger.

So if we need to find if 8 is in our tree, we'd start at 5 and then because 8 is greater than 5, we'd go right, and so on and so forth. At every decision point, we only need to explore one path.

Now if this were a normal list (or a tree, but so 'unbalanced' / degenerate as to basically be a list), it would take us more steps on average to see if 8 is in there.

## 4.1 Defining binary search trees and manually deriving *Show* for them

Now let's give a recursive definition of binary trees. A tree is:

- an empty tree, or
- a node that contains some value (its 'label') and has two sub-trees.

It's a perfect fit for a recursive data type:

```
ghci 69> data Tree a = EmptyTree | Node a (Tree a) (Tree a)
           deriving (Read, Eq)
```

We want to display trees in a readable manner, so we will manually define the *Show* instance:

```
ghci 70> instance (Show a) => Show (Tree a) where {
    show EmptyTree = "-";
    show (Node x left right) = "[" ++ show left ++
                                " " ++ show x ++ " " ++
                                show right ++ "]" }
```

## 4.2 Inserting elements in trees

Instead of manually building a tree, we're going to make a function that takes a tree and an element and inserts the element in the tree:

- we do this by comparing the value we want to insert and the root node: if the value is smaller than the root node, we go left, and if it's bigger we go right;
- we do the same for every subsequent node until we reach an empty tree;
- once we've reached an empty tree, we just insert a node with that value instead of the empty tree.

In imperative languages with mutable data structures like C, i.e., with data structures that can be destructively updated in place, we'd do this by modifying the pointers and values inside the tree.

Since Haskell is a purely functional language, once an object is created (and assigned to a name), it cannot be destructively updated: it is simply the semantic value of an expression and we have no variables and variable assignments that we 'update'.

So we can't change our tree because it is a persistent, immutable data structure:

- we have to make a new sub-tree each time we decide to go left or right in our binary search tree;
- and at the end, the insertion function returns a completely new tree;

- but this doesn't mean that the entire tree is actually duplicated in memory: there are ways to efficiently implement such manipulations of immutable data structures – see Chris Okasaki, *Purely functional data structures* (Cambridge University Press, 1999) for a detailed introduction.

Thus, the type of our insertion function is going to be  $a \rightarrow Tree\ a \rightarrow Tree\ a$ : it takes an element and a tree and returns a new tree that has that element inside.

We define two functions below:

- the first is a utility function for making a singleton tree, i.e., a tree with just one node;
- the other is the insertion function that inserts an element into a tree.

```
ghci 71> let { singleton :: a -> Tree a;
              singleton x = Node x EmptyTree EmptyTree }
```

The singleton function is just a shortcut for making a node that has a value / 'label' and then two empty sub-trees.

```
ghci 72> let { treeInsert :: (Ord a) => a -> Tree a -> Tree a;
              treeInsert x EmptyTree = singleton x;
              treeInsert x (Node r left right)
                | x == r = Node x left right
                | x < r = Node r (treeInsert x left) right
                | x > r = Node r left (treeInsert x right) }
```

In the insertion function, we first have the edge condition as a pattern: if we reached an empty sub-tree, that means we're where we want and instead of the empty tree, we put a singleton tree with our element.

If we're not inserting into an empty tree, then we have to check a couple of things:

- if the element we're inserting is equal to the root element, we just return a tree that's identical to the input tree
- if the element to be inserted is smaller than the root, we return a tree that has the same root value, the same right sub-tree but instead of its left sub-tree, it has a tree with our value inserted into it
- we do the same, but the other way around, if the value to be inserted is bigger than the root element

### 4.3 Building a tree from a list

Let's build a tree. Instead of manually building one (although we could), we'll use a fold to build up a tree from a list.

Recall that pretty much everything that traverses a list one element at a time and returns some sort of value can be implemented with a fold.

We're going to start with the empty tree and then approach the input list from the right and insert its elements one by one into our accumulator tree.

```
ghci 73> let nums = [8,6,4,1,7,3,5]
```

```
ghci 74> let numsTree = foldr treeInsert EmptyTree nums
```

In the call to *foldr*:

- *treeInsert* is the folding function (it takes a tree and a list element and produces a new tree);
- *EmptyTree* is the starting accumulator;
- *nums* is the list we're folding over.

```
ghci 75> numsTree
[[[-1-] 3 [-4-]] 5 [[-6-] 7 [-8-]]]
```

```
ghci 76> :t numsTree
numsTree :: Tree Integer
```

We see that the root node is 5 and then it has two sub-trees, one of which has 3 as its root node, while the root of the other is 7 etc.

## 4.4 Searching for elements in trees

Now we're going to make a function that checks if an element is in the tree.

We first define the edge condition: if we're looking for an element in an empty tree, then it's certainly not there.

Note how this is the same as the edge condition when we search for elements in lists: if we're looking for an element in an empty list, it's not there.

If we're looking for an element in a non-empty tree, then we have several cases:

- if the element in the root node is what we're looking for, we're done;
- if not, we can take advantage of knowing that all the left elements are smaller than the root node; so if the element we're looking for is smaller than the root node, we check to see if it's in the left sub-tree;
- if it's bigger, we check to see if it's in the right sub-tree.

All we have to do is write up this description in code, which is particularly straightforward to do in Haskell:

```
ghci 77> let { elemInTree :: (Ord a) => a -> Tree a -> Bool;
              elemInTree x EmptyTree = False;
              elemInTree x (Node r left right)
                | x == r = True
                | x < r = elemInTree x left
                | x > r = elemInTree x right }
```

Here are some examples:

```
ghci 78> 8 `elemInTree` numsTree  
True
```

```
ghci 79> 100 `elemInTree` numsTree  
False
```

```
ghci 80> 1 `elemInTree` numsTree  
True
```

```
ghci 81> 10 `elemInTree` numsTree  
False
```