Recursive Types

In this section we will consider recursively defined types, which allow us to describe structures of arbitrary size, like lists and trees. We will in particular build search trees using the mechanism of recursive types.

Reading

- Section 8.4
- Practice exercises (8.9): 3, 5

Recursive Types

A recursive type is a custom data type that refers to itself in one of its variants. In this way, a value of a particular type is either one of the basic options for that type, or a combination of simpler values, possibly of the same type.

For instance, imagine we did not have a built-in list type. We could define a list by saying:

- A value of type `List a` may be an empty list.
- A value of type `List a` may also consist of the combination ("cons") of a value of type `a` (the head) followed by a value of type `List a` (the tail).

We could write this using a custom data type declaration as follows:

```
data List a = Empty | Cons a (List a)
```

So for instance here are analogs of the lists `[0]` and `[1,2]`:

```
Cons 0 Empty
Cons 1 (Cons 2 Empty)
```

In fact we could essentially define the built-in lists as follows (`[] a` is basically the same as `[a]`):

```
data [] a = [] | (:) a [a]
```

As another example, imagine a type meant to represent arithmetic expressions. We could define such a type with something like this:

```
data Expr = Numb Double            -- A number (we only handle double precision numbers)
          | Var String             -- A (lowercase) variable (x, y, xy, err, etc)
          | Parens Expr            -- A parenthesized expression
          | Func String Expr       -- A function call (e.g. sin x)
          | Binop Expr Char Expr   -- A binary operator between two expressions
```

Then the expression `x+3` would be represented as `Binop (Var "x") "+" (Numb 3)`. Notice how the last three cases all refer to the type `Expr` itself.

Binary Search Trees

We will now use recursive types to implement binary search trees. Recall that a binary search tree is a binary tree, so each node has two (possibly empty) children, and each node also contains a value. In a binary search tree all values within the left child are less than the value at the node, and all values within the right child are greater than the value at the node.

Practice: Draw at least 3 different binary search trees consisting of the numbers 4, 6, 10, 23, 40.

We will represent a tree via a custom data type, with two variants: One representing the "empty" node and one representing an actual value node with a value and two children:

```
data Tree a = E | N (Tree a) a (Tree a)
```

We can now build trees recursively by using these constructors `E` and `N`. For example, let us start with a simple function `singleton` that turns a single value into a node containing that value and with empty children:

```
singleton :: a -> Tree a
singleton v = N E v E
```

Let us also write a function that tests if a "tree" is a "leaf". A leaf is a tree both of whose children are empty:

```
isLeaf :: Tree a -> Bool
isLeaf (N E _ E) = True
isLeaf _         = False
```

Finally, let's write a function that turns a binary tree into a list:

```
toList :: Tree a -> [a]
toList E              = []
toList N left v right = (toList left) ++ (v :: toList right)
```

We will later see other lists like this.

Now let us proceed to write an `insert` method, that inserts a new element into the proper place in the tree. It would need to have type: `Ord a => Tree a -> a -> Tree a`, so it takes an element and a tree, and returns a new tree with the element inserted in the correct spot. This will have various cases:

- If we are dealing with a normal node, compare the value at the node with the given value. If they are equal, then the number is already there and does not need to be inserted again. If the searched value is smaller than the one in the node, then we try to insert in the left child, else we insert in the right child.
- If we are dealing with an empty node, then we just form a new element.

```
insert :: Ord a => Tree a -> a -> Tree a
insert E v                = N E v E  -- Could also do as: insert E = singleton
insert (N left v' right) v
            | v == v'     = N left v' right
            | v < v'      = N (insert left v) v' right
            | v > v'      = N left v' (insert right v)
```

Practice

1. Write a function `contains :: Ord a => a -> Tree a -> Bool` that given a tree and element searches for that element in the tree.

2. Write a function `any :: (a -> Bool) -> Tree a -> Bool` that given a tree and a predicate returns `True` if there is at least one element in the tree for which the predicate is `True`, and `False` otherwise (including empty trees). Do not use ordering.

3. Write a function `all :: (a -> Bool) -> Tree a -> Bool` that given a tree and a predicate returns `True` if for all elements in the tree the predicate is `True`, and `False` otherwise. It should be `True` for empty trees (there is no element that can make the predicate `False`). Do not use ordering.

4. Write a function `min :: Ord a => Tree a -> a` that given a binary search tree finds the smallest element. It should error on an empty tree. You would more or less have to traverse the left children. Draw some tree examples before attempting this.

5. Write a function `deleteMin :: Ord a => Tree a -> a` that given a binary search tree removes the smallest element. It should error on an empty tree. You would more or less have to traverse the left children. Draw some tree examples before attempting this.