CEng 242 Homework 3

Due: 17^{th} April 2005

In this homework you will implement an abstract data type `NTree` for N-ary trees in Haskell. You will implement interface functions to manipulate this data type. In addition, you will required operator instances so that `NTree` will be member of `Ord` classes `Show`. We will not give you the type signature of `NTree`. You will implement it anyway you want and hide the type internals. That's the spirit of abstract data type as you shall remember.

Your module should look like:

```
module Nary (NTree,  interface functions) where
   data NTree alpha =  internal definition of type
    definitions of interface and auxiliary functions
```

`interface functions` are the interface functions the module will export. These functions are explained below. *`internal definitions of type`* can be anything usefull. You will not export any type constructor given here.
A node of `NTree` α stores a value of type α and can have arbitrary number of children each of which is a `NTree`. No children can be null but a leaf node can have 0 children.
You will implement the following interface functions:

`new` *`value`*
   `new ::` α `->` `NTree` α
   Creates a single, leaf node `NTree`. Node contains the *`value`*

`mergechildren` *`value childlist`*
   `mergechildren ::` α `->` `[NTree` α`]` `->` `NTree` α
   Gets a node value and a list of children and merges into a tree where root element will have *`value`* and elements in the *`childlist`* will be the children from leftmost to right in the order they are given in the list.

`getSub` *`ntree path`*
   `get ::` `NTree` α `-> [Int] -> Maybe NTree` α
   Given a tree *`ntree`* and a list of integer, this function traverses the tree and returns the corresponding subtree. Each number in the list is the number of the child node to be chosen. For example `[1,3,2]` chooses the leftmost child of the root, then the third child of that, and then the second child of this third child. An empty list simply chooses root, the whole tree.

   When such a child does not exists, for example number of children in a node is 4 and the number given in *`path`* is greater than 4 or 0, function will return `Nothing`. Otherwise it will return `Just` *`tree`* where *tree* is the tree specified in the path.

`getNode` *`ntree path`*
   `getNode ::` `NTree` α `-> [Int] -> Maybe` α
   Similar to `get` but returns the value in the node instead of the whole subtree.

`setSub` *`ntree path newtree`*
   `setSub ::` `NTree` α `-> [Int] ->` `NTree` α `->` `NTree` *alpha*
   Traverses the tree in the same way `get` does and replaces the found node with the

newtree. If the last item in the *path* is greater than the number of children in current node, *newtree* is added as the last child of the current node. Similarly if the current node is a leaf node and last item in *path* is a positive integer, a child is created. In all other cases (a value less than 0 in path, there are 2 or more elements in the path where current node is a leaf node), function will silently ignore the input and return the original tree.

`setNode` *ntree path newvalue*

`setSub :: NTree` α `-> [Int] ->` α `-> NTree` *alpha*

Similar to `setSub` but modifies the traversed node value content and do not change the others nodes and structures. The node specified in *path* should be exact (as it is in `getSub`). Otherwise no change is made and the original tree is returned.

`(<) , (==), (>), (<=), (>=), (/=)`

All of these operators are inherited from `Ord` and `Eq`. `NTree` will be implemented as an instance of `Ord` class. These operators have one of the type signature:

`Eq` α `=>` α `->` α `-> Bool`

`Ord` α `=>` α `->` α `-> Bool`

In your `instance` declarations 'α' is replaced by '`NTree` α'. Note that α should be `Ord` class in `NTree` α. So you will implement:

`instance Eq` *alpha* `=> Eq NTree` α , and

`instance Ord` *alpha* `=> Ord NTree` α

class instances. `compare, min, max` functions are also provided by standart Haskell library.

Assume the ordering relation between two trees are defined as follows:

1. If the root node of tree A is less than root node of tree B, tree A is less than tree B.
2. When the node values equal, tree A is a leaf node and tree B is a leaf node, tree A is equal to tree B
3. When the node values equal, tree A is a leaf node and tree B has at least one child, tree A is less than tree B
4. When the node values equal, and both trees have children, the leftmost children are compared and if leftmost children of A is less than the leftmost children of B, tree A is less than tree B
5. Otherwise, if leftmost children are equal, comparison continues with the next leftmost children.

In other words, the most significant value is the node content, than the children significance goes from left to right, and a null tree is always less than a non-null tree. Two trees are equal if they have exactly same structure and same corresponding node values.

Thanks to `Prelude` of Haskell, you only need to implement one of `(==)` or `(/=)` for `Eq` in addition to one of `(<=)` or `compare` for `Ord`. All remaining operators and functions come from `Ord` class definition.

`show` *tree*

`show` is inherited from `Show` class. You will implement `NTree` α as an instance of this class. Type signature of `show` is given as:

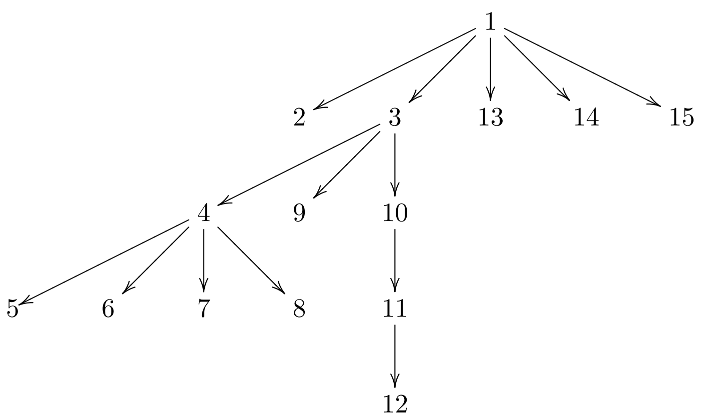
`show :: Show` α `=>` α `-> String`

You will implement the instance as:
`instance Show α => Show NTree α`

The output string will be in Lisp-like syntax with some indentation like:

```
(1,
    (2)
    (3,
        (4,
            (5)
            (6)
            (7)
            (8)
        )
        (9)
        (10,
            (11,
                (12)
            )
        )
    )
    (13)
    (14)
    (15)
)
```

Corresponding tree is:



So your module should look like:

```
module Nary (NTree, new, mergechildren, getSub, getNode, setSub, setNode,
              (<),(==),(/=),(>),(<=),(>=),compare,min,max,show ) where
   data NTree alpha =  internal definition of type
    definitions of interface and auxiliary functions
```

Put this module and nothing else in file `Nary.hs` and submit as your homework. You can have any auxiliary definitions under `where` in module definition as long as they are not exported outside.
We like to remind you that our cheating policy is to give 0 to all participants for all 6 previous and following homeworks.