CS 476 Homework #3 Due 10:45am on 2/16

Note: Answers to the exercises listed below should be emailed to Reed Oei `reedoei2@illinois.edu` in *typewritten form* (latex formatting preferred) by the deadline mentioned above. You should also email the Maude code for Problem 2 to `reedoei2@illinois.edu`.

1. Solve Exercise 3.2 in page Lecture 3.

2. Consider the following skeleton of a functional module for (binary) trees and (non-empty) lists of natural numbers. For your convenience, it imports the module `NAT` in the Maude prelude, which has all the usual numerical functions and predicates on numbers that you might need, as well as the Booleans and `if_then_else-fi` that are also available.

```
fmod LIST+TREE is protecting NAT .
   sorts NeList Tree .
   subsort Nat < NeList .
   subsort Nat < Tree .
   op _._ : Nat NeList -> NeList [ctor] .
   op _#_ : Tree Tree -> Tree [ctor] .

   vars N M : Nat .  vars T T1 T2 : Tree .  vars L L1 L2 : NeList .

   op _@_ : NeList NeList -> NeList .  *** list append

   *** include your equations defining  _@_ here

   op rev :  NeList -> NeList .        *** list reverse

   *** include your equations defining  rev here

   op sort :  NeList -> NeList .        *** list sort

   *** include your equations defining  sort here

   op add :  NeList -> Nat .            *** adds numbers in list

   *** include your equations defining  add here

   op trev : Tree -> Tree .              *** tree reverse

    *** include your equations defining  trev here

   op add :  Tree -> Nat .            *** adds numbers in tree

   *** include your equations for add here

   op t2l : Tree -> NeList .             *** converts tree into list

   *** include your equations defining  t2l here
```

```
  op l2t : NeList -> Tree .               *** converts lists to trees

   *** include your equations defining  l2t here
endfm
```

Note that natural numbers are a subsort of both non-empty lists and binary trees. Except for the imported module `NAT`, only the *data*, that is, the constructor terms, of this module are defined. What you are asked to do is to *define* the following functions on such data:

- `_@_` appends to non-empty lists
- `rev` reverses a non-empty list
- `sort` sorts a non-empty list
- `add` adds all the numbers in a non-empty list
- `trev` reverses a binary tree with numbers as leaves; geometrically, it returns the *mirror image* of the original tree.
- `add` adds all the numbers in the leaves of a binary tree
- `t2l` converts a tree into a (non-empty) list, preserving the left-to-right order in which the numbers appear in the tree into the left-to-right order in which they appear in the resulting list.
- `l2t` converts a (non-empty) list into a tree, preserving the left-to-right order in which the numbers appear in the list into the left-to-right order in which they appear in the resulting tree.

  Even giving this left-to-right order-preservation requirement, the `l2t` *is not completely determined* by such a requirement: more than one definition is possible (although one option seems the easiest to define and the most natural). That is, several trees may reasonably represent the same list preserving the left-to-right order of elements. You can give any definition you wish, provided the left-to-right order of elements is preserved.

Notes. (1) For defining some functions, the `if_then_else-fi` operator may be useful. (2) If needed, you can also define some auxiliary functions beyond those listed above. For example, this may be helpful to define the `sort` function. (3) It may help you in testing your functions to also test that they are consisten with each other. For example, have the following built-in equality predicate expressions should all evaluate to `true` when you replace the variables `T` and `L` by any concrete treess and non-empty lists. This means that these equality predicates, viewed as *equations*, are*algebraic laws* that should hold between the corresponding functions mentioned in each such equality predicates.

```
rev(t2l(T)) == t2l(trev(T)) .
add(t2l(T)) == add(T) .
add(L) == add(l2t(L)) .
t2l(l2t(L)) == L .
```

You can retrieve from the course web page this module as a "skeleton" on which to fill in your answers. Also, send a file with your solution module *and your test cases* to `reedoei2@illinois.edu`.