# CS 476 Homework #3 Due 10:45am on 9/12

**Note:** Answers to the exercises listed below should be emailed as a pdf file to Ben Clarage at the following address: `clarage2@illinois.edu` by the (hard) deadline mentioned above. They should be in *typewritten form* (latex formatting preferred).

1. Note that we can think of a relation $R \subseteq A \times B$ as a "nondeterministic function from $A$ to $B$." That is, given an element $a \in A$, we can think of the result of applying $R$ to $a$, let us denote it $R\{a\}$, as the set of all $b$'s such that $(a, b) \in R$. Unlike for functions, the set $R\{a\}$ may be empty, or may have more than one element.

   Note that the powerset $\mathcal{P}(B)$ allows us to view the "non-deterministic mapping" $a \mapsto R\{a\}$ as a *function* from $A$ to $\mathcal{P}(B)$. More precisely, we can define[1] $R\{\_\}$ as the function:

   $$R\{\_\} : A \ni a \mapsto \{b \in B \mid (a, b) \in R\} \in \mathcal{P}(B).$$

   But since this can be done for any relation $R \subseteq A \times B$, the mapping $R \mapsto R\{\_\}$ is then a function:

   $$\_\{\_\} : \mathcal{P}(A \times B) \ni R \mapsto R\{\_\} \in [A \to \mathcal{P}(B)].$$

   One can now ask an obvious question: are the notions of a relation $R \in \mathcal{P}(A \times B)$ and of a function $f \in [A \to \mathcal{P}(B)]$ *essentially the same*? That is, can we go *back and forth* between these two supposedly equivalent representations of a relation? But note that the idea of "going back and forth" between two equivalent representations is precisely the idea of a *bijection*.

   Prove that the function $\_\{\_\} : \mathcal{P}(A \times B) \ni R \mapsto R\{\_\} \in [A \to \mathcal{P}(B)]$ is bijective.

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
```

[1] Note that the function $R\{\_\}$ is *closely related* to the function

$$R[\_] : \mathcal{P}(A) \ni A' \mapsto \{b \in B \mid a \in A' \wedge \in (a, b) \in R\} \in \mathcal{P}(B)$$

defined in STACS, namely, by the equation: $R\{a\} = R[\{a\}]$. We are using a different notation ($R\{\_\}$ and $R[\_]$) to distinguish them.

```
    *** 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 (using the addition function in `NAT` as an auxiliary function)
- `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 (using the addition function in `NAT` as an auxiliary function)
- `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 have the right properties. One way you can test your functions this way is to use some equalities that express properties between functions as a *method to generate test cases by instantiating each equalitie's variables with concrete data elements in various ways*. For example, the following equalities should evaluate to `true` for any concrete instance (test case):

```
rev(t2l(T)) == t2l(trev(T)) .
sort(sort(Q:NeList)) == sort(Q:NeList) .
add(t2l(T)) == add(T) .
add(L) == add(l2t(L)) .
t2l(l2t(L)) == L .
```

where `_==_` is Maude's built-in equality predicate, which will evaluate to `true` if the equality holds for a concrete, ground instance of the variables `T`, `L` or `Q:NeList` and to `false` otherwise. Of course the above equations are *just examples*: you can think of other similar equalities between functions, or between a function and itself, that you can use to generate test cases in the same manner.

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 `clarage2@illinois.edu`.