Problem Set 3

1. Consider the following system module, whose purpose is to generate all permutations of a list `L` as the final states reachable by rewriting with the rules in the module the initial state `perm(L)`. Note that all functions in the module are *constructors*. In particular, `perm` is also a constructor term. This is because the permutations of `L` are not computed by "evaluating" `perm(L)` with some *equations*, but by changing instead the initial state `perm(L)` to other states by *rewrite rules*.

   You are asked to specify the rewrite rules (two rules are actually enough) that will make it the case that the final states reachable from `perm(L)` are exactly the permutations of `L`. Some sample search computations and the number of solutions you should get in each case are included for your convenience. Note that if a list has length n and all its elements are different, then there are $n!$ permutations of it.

```
*** if perm(L) is the initial state, then each final state is a permutations of L

mod PERMUTATIONS is protecting QID .
  sort List .
  subsort Qid < List .
  op nil : -> List [ctor] .
  op _;_ : List List -> List [ctor assoc id: nil] .

  op perm : List -> List [ctor] . *** perm(L) initial state, final states: all L's permutations

  var I : Qid .  vars L Q : List .


*** define here the transitions from perm(L) by some rules, so that the final
*** states reachable from perm(L) are exactly the permutations of L

endm

search perm(nil) =>! L .                    *** 1 solution
search perm('a) =>! L .                     *** 1 solution
search perm('a ; 'b) =>! L .                *** 2 solutions
search perm('a ; 'b ; 'c) =>! L .           *** 6 solutions
search perm('a ; 'b ; 'c ; 'd) =>! L .      *** 24 solutions
search perm('a ; 'b ; 'c ; 'd ; 'd) =>! L . *** 60 solutions
search perm('a ; 'b ; 'c ; 'd ; 'e) =>! L . *** 120 solutions
```

2. In this problem you are asked to define a *sorting algorithm* for lists of natural numbers, *not with equations, but with (transition) rules* that rewrite a list to another list with the same multiset of elements but "closer" to the sorted version of the list. If `L` is the initial state, there should be a *single* final state, namely, the sorted version of `L`. You then can just compute such a sorted version of `L` by typing in Maude:

```
rewrite L .
```

   However, since the passing from a list `L` to its sorted version is a *deterministic* process having a single answer, as a sanity check to test your rules, you should check that they are correct by checking that you always get a

single final state for each initial state `L`. To help you do that, some sample `search` commands have also been included.

Write your solution by specifying the (possibly conditional) rule or rules needed to sort a list in the system module below, so that for each list `L` the single final state will the its sorted version.

Note. Remark that *all operators in this module are constructors*. This is because no equations are used at all, so that all terms in the module are already in *normal form* by the (non-existent) equations. All computations are performed by the rule or rules that you are asked to specify, *not* by equations (except, perhaps, for the use made of some equations in `NAT` for checking an equational condition in a rule).

Hint. A single conditional rule is enough to solve this problem.

```
mod SORTING is
  protecting NAT .
  sort List .
  subsort Nat < List .
  op nil : -> List [ctor] .
  op _;_ : List List -> List [ctor assoc id: nil] .

  vars N M : Nat .  vars L Q : List .

  *** include here your rule or rules

endm

*** testing by search that your rule or rules are DETERMINISTIC (yield a single final result)

search 5 ; 4 ; 3 ; 2 ; 1 ; 0 =>! L .   *** SINGLE solution should be 0 ; 1 ; 2 ; 3 ; 4 ; 5
search 3 ; 4 ; 3 ; 5 ; 1 ; 0 =>! L .   *** SINGLE solution should be 0 ; 1 ; 3 ; 3 ; 4 ; 5
search 3 ; 4 ; 3 ; 5 ; 1 ; 4 =>! L .   *** SINGLE solution should be 1 ; 3 ; 3 ; 4 ; 4 ; 5
search 3 ; 4 ; 3 ; 4 ; 1 ; 4 =>! L .   *** SINGLE solution should be 1 ; 3 ; 3 ; 4 ; 4 ; 4

*** testing that your rules yield the correct result

rewrite 5 ; 4 ; 3 ; 2 ; 1 ; 0 .        *** should be 0 ; 1 ; 2 ; 3 ; 4 ; 5
rewrite 3 ; 4 ; 3 ; 5 ; 1 ; 0 .        *** should be 0 ; 1 ; 3 ; 3 ; 4 ; 5
rewrite 3 ; 4 ; 3 ; 5 ; 1 ; 4 .        *** should be 1 ; 3 ; 3 ; 4 ; 4 ; 5
rewrite 3 ; 4 ; 3 ; 4 ; 1 ; 4 .        *** should be 1 ; 3 ; 3 ; 4 ; 4 ; 4
```

3. You are asked to solve the following variant of the above sorting problem using a different representation of the natural numbers with 0 and 1 as constructors and with + as ACU constructor with 0 as unit element (also called "neutral" element when additive notation, as here, is used). The point is that in this representation of numbers you can solve the problem with a *single unconditional rule*. Furthermore, you *do not need to define any auxiliary functions or anything*: you just need to write the appropriate rewrite rule. The key point is that, in this representation of the natural numbers, you do not need to restrict the application of the sorting rule by checking a condition: the rule's lefhand side can do that.

```
mod SORTING-UNCONDITIONAL is
  sorts Nat List .
  subsort Nat < List .
  ops 0 1 : -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat [ctor assoc comm id: 0] .
  op nil : -> List [ctor] .
  op _;_ : List List -> List [ctor assoc id: nil] .

  vars N M : Nat .  vars L Q : List .
```

```
  *** include here your UNCONDITIONAL rule
endm

*** testing by search that your rule is  DETERMINISTIC (has a single final result)

search (1 + 1 + 1);(1 + 1) ; 1 ; 0 =>! L .
                   *** SINGLE solution should be 0 ; 1 ; (1 + 1);(1 + 1 + 1)

search (1 + 1 + 1);(1 + 1);(1 + 1 + 1) ; 1 ; 0 =>! L .
              *** SINGLE solution should be 0 ; 1 ; (1 + 1);(1 + 1 + 1);(1 + 1 + 1)

*** testing that your rule yields the correct result

rewrite (1 + 1 + 1);(1 + 1) ; 1 ; 0 . *** should be 0 ; 1 ; (1 + 1);(1 + 1 + 1)

rewrite (1 + 1 + 1);(1 + 1);(1 + 1 + 1) ; 1 ; 0 .
                              *** should be 0 ; 1 ; (1 + 1);(1 + 1 + 1);(1 + 1 + 1)
```

These two examples illustrate the expressiveness of *concurrent rewriting* as a general semantic framework for concurrency: the single sorting rule (conditional in the first case, and unconditional in the second representation) can be applied *in parallel* in different places of a list to achieve the *parallel sorting* of the list.