CS 476 Homework #10 Due 10:45am on 11/3

Note: Answers to the exercises listed below should be given in *typewritten form* (latex formatting preferred) by the deadline mentioned above. You should email your answers and also all the Maude code and all screenshots of your tool interactions to `nishant@illinois.edu`.

1. Consider the following module `PEANO+RxR` defining addition and multiplication of numbers in Peano notation:

```
set include BOOL off .

fmod PEANO+RxR is
    sort Nat .
    op 0 : -> Nat [ctor metadata "0"] .
    op s : Nat -> Nat [ctor metadata "1"] .
    op _+_ : Nat Nat -> Nat [metadata "2"] .
    op _*_ : Nat Nat -> Nat [metadata "3"] .

    vars N M : Nat .

    eq N + 0 = N .
    eq N + s(M) = s(N + M) .

    eq N * 0 = 0 .
    eq N * s(M) = N + (N * M) .
endfm
```

   You are asked to prove, using the NuITP, that multiplication is both left- and right-distributive over addition, i.e., the two inductive theorems:

```
Z:Nat * (X:Nat + Y:Nat) = (Z:Nat * X:Nat) + (Z:Nat * Y:Nat)

(X:Nat + Y:Nat) * Z:Nat = (X:Nat * Z:Nat) + (Y:Nat * Z:Nat)
```

   As pointed out in Lecture 16, you can use the NuITP in a *dumb* way, or in a *smart* way, requiring much fewer proof steps. Since you can use any results already proved about natural number addition, such as that + is AC as proved in Lecture 16, and since proving properties about multiplication will be much easier using the knowledge that + is AC, the first things that a smart user of the NuITP would do would be to use the program equivalence `PEANO+RxR` $\equiv_{sem}$ `PEANO+ACxR` to begin instead the proof with the equivalent program:

```
fmod PEANO+ACxR is
    sort Nat .
    op 0 : -> Nat [ctor metadata "0"] .
    op s : Nat -> Nat [ctor metadata "1"] .
    op _+_ : Nat Nat -> Nat [assoc comm metadata "2"] .
    op _*_ : Nat Nat -> Nat [metadata "3"] .

    vars N M : Nat .

    eq N + 0 = N .
    eq N + s(M) = s(N + M) .
```

```
    eq N * 0 = 0 .
    eq N * s(M) = N + (N * M) .
endfm
```

where the fact of + being AC has been *internalized*. If you have understood well enough the ideas in Lecture 16, you should be able to give a quite short proof of left- and right-distributivity: two applications of `gsi!` are enough if the appropriate choices of auxiliary properties and internalization are made. You can earn an **extra credit** of 5 extra points (that is, you can score a maximum of 15 points instead of a maximum of 10 on this problem) if you manage to prove left- and right-distributivity with just two applications of `gsi!`.

Additional Extra Credit. You can earn 5 more extra points on this problem if you can prove that multiplication is AC. Internalizing previously proved results about multiplication, this has also a short, "smart" proof: a single application of `gsi!` followed by a single application of `eq!` are enough to prove multiplication AC.

Advice. A more robust new alpha version of the NuITP (Alpha 14) is now available on the course web page. You wil minimize the chances of getting some strange behavior in the tool by using this latest version.

2. 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
```