CS 476 Homework #7 Due 10:45am on 10/13

Note: Answers to the exercises listed below in *typewritten form* (latex formatting preferred) as well as code solutions should be emailed by the above deadline to `nishant2@illinois.edu`.

1. Solve **Exercise** 11.2 in page 4 of Lecture 11.

   For Extra Credit. You can double the grade for this problem (i.e., this first problem can count twice as much, if you solve it and solve the extra credit question correctly) if you can also prove **Exercise** 11.3 in page 6 of Lecture 11.

2. Consider the following module (available in the course web page) of lists with a list append functions that is associative and has an identity, but where associativity and identity are explicitly defined by equations (it was already used in Problem 2 of Homework 7, where you were asked to check joinability of its critical pairs).

```
fmod LIST-EXAMPLE is
 sorts Element NeList List .
 subsorts Element < NeList < List .
 op a : -> Element [ctor] .
 op b : -> Element [ctor] .
 op c : -> Element [ctor] .
 op nil : -> List [ctor] .
 op _;_  : List List -> List .
 op _;_  : Element NeList -> NeList [ctor] .
 eq (L:List ; P:List) ; Q:List = L:List ; (P:List ; Q:List) .
 eq L:List ; nil = L:List .
 eq nil ; L:List = L:List .
endfm
```

   The main goal of this exercise is to make you familiar with how, with the help of various Maude-based tools, you can prove that a module satisfies all the required *executability conditions*: termination, sort-decreasingness, confluence, and sufficient completeness. Another goal of this exercise is to let you see how, thanks to subsorts, an operator like `_;_` can be *both* a constructor with the typing `_;_  : Element NeList -> NeList [ctor]` *and* a *defined symbol*, defined by the above three equations, with the typing `_;_  : List List -> List .`

   First of all you should prove that this module is *terminating* using the MTA Tool. Once you have proved termination, to prove that it is *confluent* you just need to prove that it is *locally confluent*, and for that it is enough to show that its critical pairs can be joined, which can be automatically checked by Maude's Church-Rosser Checker (CRC) Tool. Furthermore, the CRC tool also checks the *sort decreasingness* of the rules automatically at the same time that it checks local confluence. Finally, you can prove that this module is *sufficiently complete* using the SCC tool. You are asked to check that:

   - `LIST-EXAMPLE` is *terminating* using the the MTA tool (follow the link to MTA in the "Readings Material" section of the Course web page). You can try to do so using either an RPO order, or a polynomial order. Whichever works for you. You can get **extra credit** (specifically, if you prove termination using both RPO and polynomial orderings, your grade for this problem will be multiplied by a 1.5 factor) by proving it terminating in *both* ways, i.e., using an RPO order for one proof, and a polynomial order for an alternative proof. Specifically, if you prove termination using both RPO and polynomial orderings, your grade for this problem will be multiplied by a 1.5 factor.
   - `LIST-EXAMPLE` is *confluent* and *sort decreasingness* (the same command checks both properties) using the Maude Church-Rosser Checker.

- `LIST-EXAMPLE` is *sufficiently complete* using the Maude SCC Tool.

Explanation on How to Use the Tools. To install the tools you will need, see the course website. The Maude Termination Assistant and SCC tool are standalone tools (the SCC tool is not fully available in the MFE), and the Church-Rosser Checker is part of the MFE. Instructions for each tool are listed below, and example interactions can also be found on the course website.

- Load the module, ensuring you used `set include BOOL off .`
- Load the tool: `load path/to/tool.maude`
- Select the tool and run it on the loaded module:

MTA

```
(check-AvCrpo MODULE-NAME .)
--- Or
(check-poly MODULE-NAME .)
```

SCC

```
loop init-cc .
select CC-LOOP .
(scc MODULE-NAME .)
```

CrC

```
(select tool CRC .)
(ccr MODULE-NAME .)
```

You should include screenshots of your interactions with the tools as well as your meta-data annotations for your proof(s) of termination using MTA in your answers. (**For extra credit**) If you wish, you can develop another version with different `metadata` information for your alternative proof of termination with an alternative termination proof method using MTA. If you run into any problems using the tools, Nishant Rodrigues can answer your queries.

As mentioned above, a moral of this example is that subsorts and subsort overloading are very powerful, since they allow us in this example to tighten the typing of the general "list append" operator exactly where we want it to get a "cons-like" constructor operator. A second moral is that the essential distinction between "cons" and "append" as two *different* functions in the standard treatment of lists evaporates in an order-sorted setting. A third moral is that, by making "cons" a constructor and "append" a defined function, we can make the list constructor *free* (i.e., no equations ever apply to terms built only with operators and constants having the `ctor` declaration, which is the case in this example). Note that only in an order-sorted setting is it possible for the *same* overloaded operator to be a *constructor* for some typing and a *defined symbol* for another typing.