# CS 476 Homework #5 Due 10:45am on 3/3

**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 `abir2@illinois.edu`.

1. 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:

```
(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 .`

   This module is enclosed in parentheses only because you will need to enter it this way to the Church-Rosser Checker and the SCC Tool. There is no need for such parentheses otherwise: for example, when using the MTA Tool, parentheses should not be used when entering the module, although they are used to enclose MTA commands.

   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* by proving it terminating in *both* ways, i.e., using an RPO order for one proof, and a polynomial order for an alternative proof.
   - `LIST-EXAMPLE` is *confluent* and *sort decreasingness* (the same command checks both properties) using the Maude Church-Rosser Checker, which is part of the Maude Formal Environment (MFE) also available in the course web page.
   - `LIST-EXAMPLE` is *sufficiently complete* using the Maude SCC Tool,[1] which is also part of MFE.

[1] The use of the SCC Tool is explained in the CS 476 Lecture 7, pages 34–40. Those slides illustrate the use of SCC as a standalone tool, but SCC is currently part of the MFE. That is why `LIST-EXAMPLE` should be entered to the MFE inside parentheses and the user interaction with the SCC through the MFE is slightly different than that in the standalone mode shown in slides.

**Warning**. Since the `metadata` information needed for the termination proofs is *different* for each termination method and *irrelevant* for confluence and sufficient completeness, the best approach is for you to: (1) write a first version of `LIST-EXAMPLE` where you have added the `metadata` information for your chosen termination proof method; (2) (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; and (3) use just the version as given above for the interaction with the CRC and SCC tools through the MFE.

**Note:** As already mentioned, the outer parentheses around `LIST-EXAMPLE` are only needed when using the MFE. Email you code for all the checks to `abir2@illinois.edu`.

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.

2. The above exercise was a relatively easy exercise to help you become familiar with some of the Maude tools and learn how you can check key executability conditions for a functional module. In this second exercise you are asked to actually *prove* yourself that the module `LIST-EXAMPLE` in Exercise 1 is in fact *locally confluent* (and therefore confluent since, presumably you have proved it terminating in Exercise 1) by:

   - Considering all possible overlaps[2] between the three rules in the module (including overlaps of a rule with a renamed version of itself).
   - Computing for such overlaps the corresponding critical pairs.
   - Checking that each such critical pair can be joined by rewriting with the rules of `LIST-EXAMPLE`.

   Of course, to compute each critical pair you must perform a unification. You can do that by hand yourself; but it may be convenient for you to know that Maude can perform any desired order-sorted unification for you using the `unify` command (see Section 12.4 of the Maude manual).

   Another thing that Maude can do for you is to check the joinability of a critical pair $u = v$. All you need to do is to: (i) enter the module `LIST-EXAMPLE` into Maude, and (ii) for each critical pair $u = v$ whose variables should be *declared on the fly*, e.g., `L:List`, `E:Element`, etc., execute the command:

   ```
   red u == v .
   ```

   so that if you get the result `true` you have shown that the critical pair $u = v$ is *joinable*.

   If you use Maude to help you in these two ways, you should include a screenshot of the answers you get from Maude for each problem you ask Maude about.

[2] You do not need to consider *trivial overlaps of a rule with itself at the top position* $\epsilon$. For example, the rule `L:List ; nil = L:List` overlaps *at the top position* with a renamed version of itself, say, `Q:List ; nil = Q:List`, with unifier $\{\texttt{L} : \texttt{List} \mapsto \texttt{Q} : \texttt{List}\}$. But this yields the trivial critical pair $\langle \texttt{Q} : \texttt{List}, \texttt{Q} : \texttt{List} \rangle$, which is trivially joinable. However, you must consider all overlaps of a rule with a renamed version of itself at *non-top positions*. Such overlaps are not trivial and give rise to critical pairs that are not obviously joinable: their joinability has to be checked.