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:

```markdown
(fmod LIST-EXAMPLE is
  sorts Element List .
  subsorts Element < List .
  op a : -> Element [ctor] .
  op b : -> Element [ctor] .
  op c : -> Element [ctor] .
  op nil : -> List [ctor] .
  op _;_ : List List -> List .
  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)
```

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.

This module can be proved terminating by methods that will be explained later in the course. Therefore, to prove that it is confluent on just needs 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 Tool. You are asked to check that:

- **LIST-EXAMPLE** is confluent 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\(^1\) (also part of 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 fanyang6@illinois.edu.

A moral of this example is that subsorts and subsort overloading are very powerful, since they allow us in this example to tighten the append constructor exactly where we want it as a “cons-like” operator. A second moral is that the essential distinction between “cons” and “append” as two different functions 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. 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 an easy exercise to help you become familiar with some of the Maude tools and learn how you can check key executability conditions for a 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, as stated above, it can be proved terminating) by:

\(^1\)The use of the SCC Tool will be explained in the CS 476 lecture of 2/14. The rest of this homework does not depend on SCC. Therefore, you can start working on all other parts of this homework and begin using the SCC tool after the lecture in 2/14.
• 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 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:

\[
\text{red } u == v .
\]

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 \(\{L:\)List \mapsto Q:\)List\}. But this yields the trivial critical pair \(\langle Q:\)List, Q:\)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.