CS 476 Homework #2 Due 10:45am on 9/5

Note: Answers to the exercises listed below should be emailed as a pdf file to Ben Clarage at the following address: `clarage2@illinois.edu` by the (hard) deadline mentioned above. They should be in *typewritten form* (latex formatting preferred).

1. Give your solution to the following elementary set theory exercises in *Set Theory and Algebra in Computer Science – A Gentle Introduction to Mathematical Modeling* (STACS):

   - 51
   - 62
   - 65

2. The order used in diccionaries and phone books to sort words in a linear order (based on the usual linear order of letters in an alphabet) is called the *lexicographic order*. Let us denote it $>_{lex}$. For example, for the usual aphabet we have:

   $$a >_{lex} arm >_{lex} armory >_{lex} zoo >_{lex} zygote$$

   The purpose of this Problem is to let you have the fun of giving a *mathematical definition* of the lexicographic order by equations which at the same time is a functional program computing that order. In Maude you have complete freedom to define those equations in any way you like. But of course they should satisfy the **unique termination** and **sufficient completeness** requirements explained in Lecture 3, and should correctly define the lexicographic order $>_{lex}$.

   You should:

   - include your code and a screenshot of the evaluation in Maude of your test cases in your homework; and
   - email also the Maude code and the test cases to `clarage2@illinois.ed`.

   Lexicographic order can be defined for lists of anything (letters, numbers, etc.), provided we have given a linear order $\_ > \_$ on the list elements. For concreteness, and to get the usual case of dictionaries as an example, we will consider lists on the standard alphabet. We give you a module that *has everything you need* to define the desired function; so you only need to write the *right equations* defining the function `>lex`. Here is the module (**which you can retrieve from the course web page**):

```
fmod LEX is

*** declarations borrowed from module BOOL-DATA and BOOLEAN

 sort Boolean .
 ops tt ff : -> Boolean [ctor] .
 op _or_ : Boolean Boolean -> Boolean .
 op _and_ : Boolean Boolean -> Boolean .
 op not : Boolean -> Boolean .
 var B B' : Boolean .
 eq tt or B = tt .    eq tt and B = B .    eq not(tt) = ff .
 eq ff or B = B .     eq ff and B = ff .   eq not(ff) = tt .
```

```
*** definition of if-then-else for Booleans

 op if : Boolean Boolean Boolean -> Boolean .
 eq if(tt,B,B') = B .
 eq if(ff,B,B') = B' .

*** definition of letters and of order and equality between letters;
*** only some letters used to avoid longer definition

 sort Letter .
 ops a b c d e f i m : -> Letter [ctor] .
 op _>_ : Letter Letter -> Boolean .
 op _~_ : Letter Letter -> Boolean .

eq a > a = ff .  eq b > a = ff .  eq c > a = ff .  eq d > a = ff .
eq a > b = tt .  eq b > b = ff .  eq c > b = ff .  eq d > b = ff .
eq a > c = tt .  eq b > c = tt .  eq c > c = ff .  eq d > c = ff .
eq a > d = tt .  eq b > d = tt .  eq c > d = tt .  eq d > d = ff .
eq a > e = tt .  eq b > e = tt .  eq c > e = tt .  eq d > e = tt .
eq a > f = tt .  eq b > f = tt .  eq c > f = tt .  eq d > f = tt .
eq a > i = tt .  eq b > i = tt .  eq c > i = tt .  eq d > i = tt .
eq a > m = tt .  eq b > m = tt .  eq c > m = tt .  eq d > m = tt .

eq e > a = ff .  eq f > a = ff .  eq i > a = ff .  eq m > a = ff .
eq e > b = ff .  eq f > b = ff .  eq i > b = ff .  eq m > b = ff .
eq e > c = ff .  eq f > c = ff .  eq i > c = ff .  eq m > c = ff .
eq e > d = ff .  eq f > d = ff .  eq i > d = ff .  eq m > d = ff .
eq e > e = ff .  eq f > e = ff .  eq i > e = ff .  eq m > e = ff .
eq e > f = tt .  eq f > f = ff .  eq i > f = ff .  eq m > f = ff .
eq e > i = tt .  eq f > i = tt .  eq i > i = ff .  eq m > i = ff .
eq e > m = tt .  eq f > m = tt .  eq i > m = tt .  eq m > m = ff .

*** letter equality defined in terms of _>_ to avoid tedious case-by-case definition

vars X Y : Letter .
eq (X ~ Y) = not(X > Y or Y > X) .

*** words on an alphabet as lists of letters

 sort Word .
 op ! : -> Word [ctor] .              *** empty word
 op _ _ : Letter Word -> Word [ctor] .

*** lexicographic order

 op _>lex_ : Word Word -> Boolean .

**** add your mathematical variable declarations here

*** add your recursive equations defining _>lex_ here

endfm
```

You shoud **test** your module using a substantial set of **test cases**. Here are a few test cases for which you definition, if it is correct, should give the same answer:

```
Maude> red a m ! >lex a ! .
result Boolean: ff

Maude> red a ! >lex a m ! .
result Boolean: tt

Maude> red f a m e ! >lex b a d ! .
result Boolean: ff

Maude> red b a d ! >lex f a m e ! .
result Boolean: tt

Maude> red i f ! >lex c a d ! .
result Boolean: ff

Maude> red c a d ! >lex i f ! .
result Boolean: tt

Maude> red d a m ! >lex d a m e ! .
result Boolean: tt

Maude> red d a m e ! >lex d a m ! .
result Boolean: ff
```