CS 476 Homework #10 Due 10:45am on 4/14

Note: Answers to the exercises listed below and all Maude code for exercises requiring it should be emailed to `abir2@illinois.edu`.

1. In this problem you are asked to define a *sorting algorithm* for lists of natural numbers, *not with equations, but with (transition) rules* that rewrite a list to another list with the same multiset of elements but "closer" to the sorted version of the list. If `L` is the initial state, there should be a *single* final state, namely, the sorted version of `L`. You then can just compute such a sorted version of `L` by typing in Maude:

   ```
   rewrite L .
   ```

   However, since the passing from a list `L` to its sorted version is a *deterministic* process having a single answer, as a sanity check to test your rules, you should check that they are correct by checking that you always get a *single final state* for each initial state `L`. To help you do that, some sample *search* commands have also been included.

   Write your solution by specifying the (possibly conditional) rule or rules needed to sort a list in the system module below, so that for each list `L` the single final state will the its sorted version.

   Note. Remark that *all operators in this module are constructors.* This is because no equations are used at all, so that all terms in the module are already in *normal form* by the (non-existent) equations. All computations are performed by the rule or rules that you are asked to specify, *not* by equations (except, perhaps, for the use made of some equations in `NAT` for checking an equational condition in a rule).

   Hint. A single conditional rule is enough to solve this problem.

   ```
   mod SORTING is
     protecting NAT .
     sort List .
     subsort Nat < List .
     op nil : -> List [ctor] .
     op _;_ : List List -> List [ctor assoc id: nil] .

     vars N M : Nat .  vars L Q : List .

     *** include here your rule or rules

   endm

   *** testing by search that your rule or rules are DETERMINISTIC (yield a single final result)

   search 5 ; 4 ; 3 ; 2 ; 1 ; 0 =>! L .   *** SINGLE solution should be 0 ; 1 ; 2 ; 3 ; 4 ; 5
   search 3 ; 4 ; 3 ; 5 ; 1 ; 0 =>! L .   *** SINGLE solution should be 0 ; 1 ; 3 ; 3 ; 4 ; 5
   search 3 ; 4 ; 3 ; 5 ; 1 ; 4 =>! L .   *** SINGLE solution should be 1 ; 3 ; 3 ; 4 ; 4 ; 5
   search 3 ; 4 ; 3 ; 4 ; 1 ; 4 =>! L .   *** SINGLE solution should be 1 ; 3 ; 3 ; 4 ; 4 ; 4

   *** testing that your rules yield the correct result

   rewrite 5 ; 4 ; 3 ; 2 ; 1 ; 0 .        *** should be 0 ; 1 ; 2 ; 3 ; 4 ; 5
   ```

```
rewrite 3 ; 4 ; 3 ; 5 ; 1 ; 0 .        *** should be 0 ; 1 ; 3 ; 3 ; 4 ; 5
rewrite 3 ; 4 ; 3 ; 5 ; 1 ; 4 .        *** should be 1 ; 3 ; 3 ; 4 ; 4 ; 5
rewrite 3 ; 4 ; 3 ; 4 ; 1 ; 4 .        *** should be 1 ; 3 ; 3 ; 4 ; 4 ; 4
```

2. Consider the following dining philosophers example, that you can retrieve from the course web page:

```
fmod NAT/4 is
   protecting NAT .
   sort Nat/4 .
   op [_] : Nat -> Nat/4 .
   op _+_ : Nat/4 Nat/4 -> Nat/4 .
   op _*_ : Nat/4 Nat/4 -> Nat/4 .
   op p : Nat/4 -> Nat/4 .
   vars N M : Nat .
   ceq [N] = [N rem 4] if N >= 4 .
   eq [N] + [M] = [N + M] .
   eq [N] * [M] = [N * M] .
   ceq p([0]) = [N] if s(N) := 4 .
   ceq p([s(N)]) = [N] if N < 4 .
endfm

mod DIN-PHIL is
   protecting NAT/4 .
   sorts Oid Cid Attribute AttributeSet Configuration Object Msg .
   sorts Phil Mode .
   subsort Nat/4 < Oid .
   subsort Attribute < AttributeSet .
   subsort Object < Configuration .
   subsort Msg < Configuration .
   subsort Phil < Cid .

   op __ : Configuration Configuration -> Configuration
                                                 [ assoc comm id: none ] .
op _`,_ : AttributeSet AttributeSet -> AttributeSet
                                                 [ assoc comm id: null ] .
   op null : -> AttributeSet .
   op none : -> Configuration .
   op mode`:_ : Mode -> Attribute [ gather ( & ) ] .
   op holds`:_ : Configuration -> Attribute [ gather ( & ) ] .
   op <_:_|_> : Oid Cid AttributeSet -> Object .
   op Phil : -> Phil .

   ops t h e : -> Mode .
   op chop : Nat/4 Nat/4 -> Msg [comm] .
   op init : -> Configuration .
   op make-init : Nat/4 -> Configuration .

   vars N M K : Nat .
   var C : Configuration .

   ceq init = make-init([N]) if s(N) := 4 .
   ceq make-init([s(N)])
     = < [s(N)] : Phil | mode : t , holds : none >  make-init([N])  (chop([s(N)],[N]))
     if N < 4 .
   ceq make-init([0]) =
      < [0] : Phil | mode : t , holds : none > chop([0],[N]) if  s(N) := 4 .
```

```
   rl [t2h] : < [N] : Phil | mode : t , holds : none > =>
      < [N] : Phil | mode : h , holds : none > .
   crl [pickl] :  < [N] : Phil | mode : h , holds : none > chop([N],[M])
       => < [N] : Phil | mode : h , holds :  chop([N],[M]) > if [M] = [s(N)] .
   rl [pickr] :  < [N] : Phil | mode : h , holds : chop([N],[M]) >
      chop([N],[K]) =>
      < [N] : Phil | mode : h , holds :  chop([N],[M]) chop([N],[K]) > .
   rl [h2e] :  < [N] : Phil | mode : h , holds :  chop([N],[M])
      chop([N],[K]) > => < [N] : Phil | mode : e ,
      holds :  chop([N],[M]) chop([N],[K]) > .
   rl [e2t] :  < [N] : Phil | mode : e , holds :  chop([N],[M])
      chop([N],[K]) >  =>   chop([N],[M]) chop([N],[K])
      < [N] : Phil | mode : t , holds :  none > .
endm
```

There are four philosophers, that you can imagine eating in a circular table. Initially they are all in thinking mode (`t`), but they can go into hungry mode (`h`), and after picking the left and right chopsticks (they eat Chinese food) into eating mode (`e`), and then can return to thinking.

The identities of the philosophers are naturals modulo 4, with contiguous philosophers arranged in increasing order from left to right (but wrapping around to 0 at 4). The chopsticks are numbered, with each chopstick indicating the two philosophers next to it.

Prove, by giving appropriate search commands from the initial state `init`, the following properties:

- (contiguous mutual exclusion): it is never the case that two *contiguous* philosophers are eating simultaneously.
- (mutual non-exclusion): it is however possible for two philosophers to eat simultaneously.
- (three exclusion): it is impossible for three philosophers to eat simultaneously.
- (deadlock) the system can deadlock.