CS 476 Homework #10 Due 10:45am on 10/31

Note: Answers to the exercises listed below and all Maude code and screenshots of tool interactions should be emailed to `clarage2@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
```

For Extra Credit. You can earn 50% extra credit in Problem 1 if you correctly solve the following variant of the above sorting problem using a different representation of the natural numbers with 0 and 1 as constructors and with + as ACU constructor with 0 as unit element (also called "neutral" element when additive notation, as here, is used). The point is that in this representation of numbers you can solve the problem with a *single unconditional rule*. Furthermore, you *do not need to define any auxiliary functions or anything*: you just need to write the appropriate rewrite rule. The key point is that, in this representation of the natural numbers, you do not need to restrict the application of the sorting rule by checking a condition: the rule's lefhand side can do that thanks to the remarkable expressive power of rewriting modulo *ACU*.

```
mod SORTING-UNCONDITIONAL is
  sorts Nat List .
  subsort Nat < List .
  ops 0 1 : -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat [ctor assoc comm id: 0] .
  op nil : -> List [ctor] .
  op _;_ : List List -> List [ctor assoc id: nil] .

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

  *** include here your UNCONDITIONAL rule
endm

*** testing by search that your rule is  DETERMINISTIC (has a single final result)

search (1 + 1 + 1);(1 + 1) ; 1 ; 0 =>! L .
                    *** SINGLE solution should be 0 ; 1 ; (1 + 1);(1 + 1 + 1)

search (1 + 1 + 1);(1 + 1);(1 + 1 + 1) ; 1 ; 0 =>! L .
              *** SINGLE solution should be 0 ; 1 ; (1 + 1);(1 + 1 + 1);(1 + 1 + 1)

*** testing that your rule yields the correct result

rewrite (1 + 1 + 1);(1 + 1) ; 1 ; 0 . *** should be 0 ; 1 ; (1 + 1);(1 + 1 + 1)

rewrite (1 + 1 + 1);(1 + 1);(1 + 1 + 1) ; 1 ; 0 .
                              *** should be 0 ; 1 ; (1 + 1);(1 + 1 + 1);(1 + 1 + 1)
```

These two, closely related examples illustrate the expressiveness of *concurrent rewriting* as a general semantic framework for concurrency: the single sorting rule (conditional in the first case, and unconditional in the second representation) can be applied *in parallel* in different places of a list to achieve the *parallel sorting* of the list.

2. Consider the following example. It is a desugared version of an object-oriented specification of a dining philosophers protocol that you can retrieve from the course web page:

```
fmod NAT/4 is
   protecting NAT .
   sort Nat/4 .
   op [_] : Nat -> Nat/4 .
   op p : Nat/4 -> Nat/4 .
   vars N M : Nat .
   ceq [N] = [N rem 4] if N >= 4 .
   eq p([0]) = [3] .
   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 .
   vars C C1 C2 C3 : 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 natural numbers modulo 4. Such numbers, as well as a predecessor function `p` on them, are defined in the `NAT/4` functional module. In the circular table contiguous philosophers are 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. Therefore, when no philosopher has yet picked up a chopsitick, each philosopher will have a chopstick on his/her right and another on his/her left.

Of course, all this is a metaphor due to Edsger Dijsktra: philosophers model processes and chopsticks model resources to be safely shared among them, and the metaphor describes how a flexible form of mutual exclusion can be achieved.

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

(a) (contiguous mutual exclusion): it is never the case that two *contiguous* philosophers are eating simultaneously.

(b) (mutual non-exclusion): it is however possible for two philosophers to eat simultaneously.

(c) (three exclusion): it is impossible for three philosophers to eat simultaneously.

(d) (deadlock) the system can deadlock.

Deadlock is a *bad* property for any mutual exclusion protocol, since such protocols should never terminate. There are of course deadlock-free versions of a dining philosophers protocol. The point of this version is to illustrate that a given protocol *design* may not be what we want; but we can find out about both its good properties and its design errors by model cheking: here properties (a)–(c) are good ones, since conflicting access to resources (holding the same chopstick) is avoided and yet several processes can use such resources simultaneously (here two processes at a time); but the model checking analysis uncovers a design flaw, namely, the lack of deadlock freedom, which is property (d).

For Extra Credit. Call a search command *constrained* if it has the form:

```
search init =># u s.t. C .
```

where `u` is a constructor pattern, `C` is a constraint, i.e., condition, and `#` can be `1`, `+`, `*` or `!`. That is, the command uses the constrained constructor pattern $u \mid C$. Otherwise, the search command is called *unconstrained.* Of course, unconstrained search commands are simpler and more efficient. You can get 50% extra credit on Problem 3 if you can verify all properties (a)–(d) using three unconstrained search commands and only one constrained search command.