CS 476 Homework #2 Due 10:45am on 9/24

Note: Answers to the exercises listed below should be emailed as a pdf file to the instructor at the following address: `meseguer@illinois.edu` by the (hard) deadline mentioned above. They should be in pdf and in *typewritten form* (latex formatting preferred). For exercises 1 and 2 you should include your code and the results of evaluating your test cases in the pdf text of your homework. Furthermore, you should also attach (as separate files) the Maude files for Problems 1 and 2 to the email containing your answers to the homework.

Note: Each exercise is evaluated from 0 to 10. If you do the three of them perfectly you get a 30 mark in the homework. Furthermore, as explained below, you can get some extra credit for Exercise 3.

1. Consider the following skeleton of a Maude functional module defining the integers and several functions on them:

```
fmod INTEGER is  protecting BOOL .
  sorts Zero NzNat Nat NzNeg Neg Int .
  subsorts Zero NzNat < Nat < Int .
  subsorts Zero NzNeg < Neg < Int .
  op 0 : -> Zero [ctor] .
  op s : Nat -> NzNat [ctor] .
  op p : Neg -> NzNeg [ctor] .
  ops s p : Int -> Int .
  op - : Int -> Int .
  op _+_ : Int Int -> Int .
  op _*_ : Int Int -> Int .
  op _>_ : Int Int -> Bool .

  vars i j : Int .  vars n m : Nat .  vars n' m' : NzNat .
  vars q r : Neg .  vars q' r' : NzNeg .

  *** add here your equations defining s, p, -, _+_, _*_ and _>_

endfm
```

Note the very "symmetric" nature of the integers in this order-sorted definition. The sort `Zero` with constructor constant `0` is both a subsort of `Nat` and of the negative numbers `Neg`. The strictly positive (resp. strictly negative) numbers are in the subsort `NzNat` (resp. `NzNeg`). The successor `s` (resp. predecessor `p`) constructors build the data elements in `NzNat` (resp. `NzNeg`) in a completely "symmetric" way: each constructor is the "mirror image" of the other, and so are the sorts `NzNat` and `NzNeg` and, likewise, `Nat` and `Neg`. A quite interesting fact is that `s` and `p` are also overloaded as *defined functions on the integers*, which are defined by corresponding equations (that you are asked to give) defining their meaning for any integers of sort `Int`. All the other functions are as you would expect: `-(i)` is the unary minus operator for integer `i`. We do not need a binary minus, since we could define `i - j = i + -(j)`. The operations `_+_` and `_*_` are of course integer addition and multiplication, and `_>_` is the strict order on integers.

You shoud **test** your module using a collection of **test cases**. For your convenience, several test cases are given below with the `red` command, as well as the answers that you should get. The test cases are:

```
red s(p(p(0))) .
red p(s(s(0))) .
red -(s(s(s(0)))) .
red -(p(p(p(0)))) .
red s(s(s(0))) + s(s(0)) .
red s(s(s(0))) + p(p(0)) .
red p(p(p(0))) + s(s(0)) .
red p(p(p(0))) + p(p(0)) .
red s(s(s(0))) * s(s(0)) .
red s(s(s(0))) * p(p(0)) .
red p(p(p(0))) * s(s(0)) .
red p(p(p(0))) * p(p(0)) .
red s(s(s(0))) > s(s(0)) .
red s(s(s(0))) > p(p(0)) .
red p(p(p(0))) > s(s(0)) .
red p(p(p(0))) > p(p(0)) .
```

and the corresponding evaluations are:

```
==========================================
reduce in INTEGER : s(p(p(0))) .
rewrites: 1 in 0ms cpu (0ms real) (1000000 rewrites/second)
result NzNeg: p(0)
==========================================
reduce in INTEGER : p(s(s(0))) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: s(0)
==========================================
reduce in INTEGER : -(s(s(s(0)))) .
rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNeg: p(p(p(0)))
==========================================
reduce in INTEGER : -(p(p(p(0)))) .
rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: s(s(s(0)))
==========================================
reduce in INTEGER : s(s(s(0))) + s(s(0)) .
rewrites: 3 in 0ms cpu (0ms real) (1000000 rewrites/second)
result NzNat: s(s(s(s(s(0)))))
==========================================
reduce in INTEGER : s(s(s(0))) + p(p(0)) .
rewrites: 5 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: s(0)
==========================================
reduce in INTEGER : p(p(p(0))) + s(s(0)) .
rewrites: 5 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNeg: p(0)
==========================================
reduce in INTEGER : p(p(p(0))) + p(p(0)) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNeg: p(p(p(p(p(0)))))
==========================================
reduce in INTEGER : s(s(s(0))) * s(s(0)) .
rewrites: 8 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: s(s(s(s(s(s(0))))))
==========================================
reduce in INTEGER : s(s(s(0))) * p(p(0)) .
rewrites: 16 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNeg: p(p(p(p(p(p(0))))))
==========================================
reduce in INTEGER : p(p(p(0))) * s(s(0)) .
rewrites: 8 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNeg: p(p(p(p(p(p(0))))))
==========================================
reduce in INTEGER : p(p(p(0))) * p(p(0)) .
rewrites: 16 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: s(s(s(s(s(s(0))))))
==========================================
reduce in INTEGER : s(s(s(0))) > s(s(0)) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in INTEGER : s(s(s(0))) > p(p(0)) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in INTEGER : p(p(p(0))) > s(s(0)) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: false
==========================================
reduce in INTEGER : p(p(p(0))) > p(p(0)) .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: false
```

2. Consider the following skeleton of a functional module defining several functions on lists of natural numbers:

```
fmod LIST-FUNCTIONS is protecting NAT .
  sort NeList List .
  subsort Nat < NeList < List .
  op nil : -> List [ctor] .
  op _ _ : List List -> List [ctor assoc id: nil] .
  op _ _ : NeList NeList -> NeList [ctor assoc id: nil] .
  op rev : List -> List .
  op remove_from_ : Nat List -> Nat .
  op no-reps : List -> List .
  op pal : List -> Bool .
  op prefix : List List -> Bool .

  vars n m : Nat .  vars L L1 L2 : List .  vars P Q : NeList .

  *** add here your equations defining rev, remove_from_, no-reps
  *** pal, and prefix

endfm
```

To make your life easier, the Maude built-in module `NAT` is included, so that you can use a shorter, more readable representation of lists of numbers in decimal notation, such as, e.g., a list of seven numbers of the form `1 2 3 4 5 13 271`. `NAT` includes the `BOOL` submodule (with constants `true` and `false`) as well. Both are described in *All About Maude* and in the Maude Manual; but all you may need to use are the following two built-in functions provided by `NAT`, which can be helpful for you to define some of the above functions, namely: (i) an "if-then-else" function with syntax `if_then_else_fi` and an *equality predicate* on natural numbers with syntax `_==_` so that `n == m` evaluates to `true` when numbers `n` and `m` are equal, and to `false` when they are different.

The functions that you are asked to define are as follows: `rev` reverses a list; `remove n from L` removes all occurrences of number `n` from list `L`; `no-reps(L)` is the list obtained by keeping the first occurrence of each number `n` appearing in the list and removing all other subsequent occurrences of `n` (if any); in particular, the list `no-reps(L)` will never have repeated elements; `pal(L)` is `true` if `L` is a *palindrome*, i.e., if it reads the same forwards and backwards, or, equivalently, if `L` satisfies the equation `L = rev(L)`; otherwise, `pal(L)` is `false`. The predicate `prefix(L,L1)` is `true` if `L1` can be decomposed as a list concatenation `L1 = L L2`, where `L2` could be `nil`; and is `false` otherwise.

You shoud **test** your module using a collection of **test cases**. For your convenience, several test cases are given below with the `red` command, as well as the answers that you should get. The test cases are:

```
red rev(1 2 3 4) .
red rev(1) .
red remove 3 from 1 2 3 1 3 2 3 .
red no-reps(1 2 1 3 2 3 4 3 1) .
red pal(1 2 3 4 3 2 1) .
red 1 2 3 4 3 2 1 == rev(1 2 3 4 3 2 1) .
red pal(1 2 3 4 2 1) .
red 1 2 3 4 2 1 == rev(1 2 3 4 2 1) .
red prefix(1 2,1 2 3) .
red prefix(1 2 3,1 2 3) .
red prefix(1 2 3 4,1 3 4 4) .
```

and the corresponding evaluations are:

```
==========================================
reduce in LIST-FUNCTIONS : rev(1 2 3 4) .
rewrites: 5 in 0ms cpu (0ms real) (5000000 rewrites/second)
result NeList: 4 3 2 1
==========================================
reduce in LIST-FUNCTIONS : rev(1) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 1
==========================================
reduce in LIST-FUNCTIONS : remove 3 from 1 2 3 1 3 2 3 .
rewrites: 22 in 0ms cpu (0ms real) (~ rewrites/second)
result NeList: 1 2 1 2
==========================================
reduce in LIST-FUNCTIONS : no-reps(1 2 1 3 2 3 4 3 1) .
rewrites: 57 in 0ms cpu (0ms real) (2850000 rewrites/second)
result NeList: 1 2 3 4
==========================================
reduce in LIST-FUNCTIONS : pal(1 2 3 4 3 2 1) .
rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in LIST-FUNCTIONS : 1 2 3 4 3 2 1 == rev(1 2 3 4 3 2 1) .
rewrites: 9 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in LIST-FUNCTIONS : pal(1 2 3 4 2 1) .
rewrites: 9 in 0ms cpu (0ms real) (~ rewrites/second)
```

```
result Bool: false
==========================================
reduce in LIST-FUNCTIONS : 1 2 3 4 2 1 == rev(1 2 3 4 2 1) .
rewrites: 8 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: false
==========================================
reduce in LIST-FUNCTIONS : prefix(1 2, 1 2 3) .
rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in LIST-FUNCTIONS : prefix(1 2 3, 1 2 3) .
rewrites: 10 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in LIST-FUNCTIONS : prefix(1 2 3 4, 1 3 4 4) .
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: false
```

3. A finite, unsorted signature Σ can be represented as a family of finite sets $\Sigma = \{F_n\}_{0 \leq n \leq k}$ for some $k \geq 0$, where F_0 is the set of constant symbols, F_1 is the set of unary function symbols, and F_n the set of function symbols of $n \leq k$ arguments. This problem has two parts.

   (a) (You can get up to 5 points for this part). For the Peano signature $\Sigma_{Peano} = \{F_n\}_{0 \leq n \leq 1}$ with constant $F_0 = \{0\}$ and unary symbol $F_1 = \{s\}$, given the set $A = \{a, b, c\}$, how many *different* Σ_{Peano}-algebras $\mathbf{A} = (A, \_\mathbf{A})$ can be defined on the set A?

   (b) **Extra Credit** (You can get up to 10 points for this part). Given a finite signature $\Sigma = \{F_n\}_{0 \leq n \leq k}$ for some $k \geq 0$, and a finite set A of m elements, written $|A| = m$, give a formula dependent on m and on the numbers $|F_n| = ops_n$, $0 \leq n \leq k$, computing the number of *different* Σ-algebras $\mathbf{A} = (A, \_\mathbf{A})$ that can be defined on the set A.