# CS 476 Homework #5 Due 10:45am on 4/1

**Note:** Answers to the exercises listed below should be handed to the instructor *in hardcopy* and in *typewritten form* (latex formatting preferred) by the deadline mentioned above. You should also email to (`meseguer@cs.uiuc.edu`) the Maude code for the exercises requiring that. Make sure that you use the *latest* version of the lecture notes for exercises belonging to a particular lecture.

1. Consider the following module defining the cardinality of multisets, that you can download from the course web page:

```
fmod CARD is
  sorts Natural MSet .
  subsort Natural < MSet .
  op 0 : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  op _+_ : Natural Natural -> Natural .
  op __ : MSet MSet -> MSet [ctor assoc comm] .
  op card : MSet -> Natural .
  vars N M : Natural .
  vars U V : MSet .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
  eq card(N U) = s(card(U)) .
endfm
```

   Do the following:

   - Use the Maude Sufficient Completeness Checker (SCC) to show that this module is not sufficiently complete. Use the information given by the SCC to complete the module, giving the most natural possible definition of cardinality in the missing case so as to cover all cases.
   - Use the SCC again to check that your corrected definition is now sufficiently complete.
   - Use the ITP to state and prove that the following equality holds inductively in your corrected module:

     ```
     ((card(U:MSet V:MSet)) = (card(U:MSet) + card(V:MSet)))
     ```

     Note the fact that both the above equation as a whole, as well as the left side and right side of the equation, are enclosed in parentheses. This is to help the, somewhat primitive, parser of the ITP, which may not be able to parse a goal unless the equation in the goal has enough parentheses.

2. Consider the definition of binary trees with quoted identifiers on the leaves given in Lecture 13. Complete the module given below (available in the course web page) by defining with confluent and terminating equations the two functions called `leaves` and `inner`, that count, respectively, the number of leaf nodes of a tree, and the number of nodes in a tree that are *not* leaf nodes. For example, for the tree `(('a # 'b) # 'c) # 'd` there are 4 leaf nodes (namely `'a`, `'b`, `'c`, and `'d`), and 3 inner nodes (corresponding to the 3 different occurrences of the `#` operator).

```
fmod TREE is
  protecting QID .
```

```
  sorts Natural Tree .
  subsort Qid < Tree .
  op 0 : -> Natural [ctor].
  op s : Natural -> Natural [ctor].
  op _+_ : Natural Natural -> Natural [assoc comm].
  op _#_ : Tree Tree -> Tree [ctor] .
  ops leaves inner : Tree -> Natural .
  var I : Qid .
  vars N M : Natural .
  vars T T' : Tree .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
  *** add equations for leaves and inner here
endfm
```

Once you have defined and tested your definitions for `leaves` and `inner` do the following, *including screenshots for each tool used* in your hardcopy of the homework:

- check that it is sufficiently complete using the SCC tool
- state a theorem, in the form of a universally quantified equation, that gives a general law stating, for any tree `T`, the exact relation between the numbers `leaves(T)` and `inner(T)`
- give a mechanical proof of that theorem using the ITP

3. The following two exercises use a *special* version (with support for this Java subset) of the ITP downloadable from the course web site, which has an extension of the list of commands of the ITP specifically designed to support Hoare logic reasoning in our programming language. This latest version should only be used for verifying Java programs. For all other purposes, for example to verify inductive properties of a functional module, you should still use the *previous* version of the ITP that you have used before. For any questions/problems that you may have using this experimental version of the Java-ITP tool you can consult Ralf Sasse (`rsasse@cs.uiuc.edu`).

   First unpack the downloaded file with 'tar xvfz homework-cs477.tar.gz'

   The way this extension works is as follows:

   - start Maude 2.1.1 (the tool needs exactly this version)
   - load into Maude the functional module defining the semantics of our language, `java-es-flat.maude`, (for speed reasons we use the flattened version).
   - define a program (`BlockStatements`) `foo` in a module `foo.maude` importing `java-es-flat.maude` by declaring a constant `foo` and giving an equation defining the constant as the corresponding program text. Also declare a constant `foo-init` which contains all declarations necessary for your program and make sure there are no ("Java") declarations in your program `foo`. The module `foo.maude` should also contain all *auxiliary functions* needed in a proof of correctness of program `foo`. (Note that in the two excercises below this has already been done for you, in the files div2.maude and sumn.maude.)
   - load `foo.maude` (The above mentioned file instead)
   - then load the included version of `itp-tool.maude`
   - then, to prove a Hoare triple
     $$\{P\}\ \texttt{foo}\ \{Q\}$$
     you give a `javax` command, or a `javax-inv` command which also needs an invariant.

   In the homework this command has already been spelled out for you in "sumn.itp" whereas in "div2.itp" the invariant is missing. Before loading "div2.itp" you need to enter your own invariant in the marked place. Then you can interact with the ITP tool to discharge the created first-order goals.

   Note that you can write your solution (i.e. the commands you used to discharge a goal) directly below the javax-inv command and then in one shot load the module which will create the goal and discharge it.

   Here is the sumn.maude module, that you can download es explained above:

```
fmod SUMN-JAVAX is
including JAVAX .
op sum : Int -> Int .
var N : Int .
ceq sum(N) = 0 if N <= 0 .
ceq sum(N) = N + sum(N - 1) if 0 < N .

op sumn : -> BlockStatements .
op sumn-init : -> BlockStatements .
eq sumn =
   'C = #i(1) ; 'X = #i(0) ;
   while ('C <= 'N)
     { 'X = 'X + 'C ;
       'C = 'C + #i(1) ;
   } .
eq sumn-init = (int 'C ; int 'X ; int 'N ;) .
endfm
```

and here is the sumn.itp goal you have to prove (downloadable the same way):

```
select ITP-TOOL .
loop init-itp .

(javax-inv SUMN-JAVAX :
--- specification variables
(N:Int)
--- precondition
(
((int-val(S:WrappedState['N])) = (N:Int)
& (0 <= N:Int) = (true))
 )
--- program
sumn-init
sumn
--- postcondition
(
(int-val(S:WrappedState['X]))
= (sum(N:Int))
)
--- invariant
(
(int-val(S:WrappedState['X]))
= (sum(int-val(S:WrappedState['C]) + -(1)))
&
(1 <= int-val(S:WrappedState['C]))
= (true)
&
((int-val(S:WrappedState['C]) + -(1)) <=
 int-val(S:WrappedState['N]))
= (true)
&
(int-val(S:WrappedState['N]))
= (N:Int))
.)
```

4. Similarly, here is the div2.maude module:

```
fmod DIV2-JAVAX is
including JAVAX .
op div2 : -> BlockStatements .
op div2-init : -> BlockStatements .
eq div2 = 'Y = #i(0) ;
          while (#i(1) < 'X)
            { 'X = 'X - #i(2) ;
              'Y = 'Y + #i(1) ; } .
eq div2-init = (int 'X ; int 'Y ;) .
endfm
```

and here is the goal you should prove, now with the invariant left unspecified; so you have to first come up wiht it, and then prove the goal.

```
select ITP-TOOL .
loop init-itp .

(javax-inv DIV2-JAVAX :
--- specification constants
(N:Int)
--- precondition
(((S:WrappedState['X]) = (int(N:Int)))
 & ((N:Int >= 0) = (true)))
--- program
div2-init
div2
--- postcondition
((((2 * int-val(S:WrappedState['Y]))
 + int-val(S:WrappedState['X])) = (N:Int))
& ((1 >= int-val(S:WrappedState['X])) = (true))
& ((int-val(S:WrappedState['X]) >= 0) = (true)))
--- invariant
(YOUR INVARIANT HERE)
.)
```