Comp 317: Semantics of Programming Languages

Problem Sheet 7

1. Simplify the following (check your answers using Maude):

   (a) `initial ; 'x := 2 ; if 'y < 'x then 'y := 'y + 1 else skip endif`
   `initial ; 'x := 2 ; 'y := 'y + 1`

   (b) `initial ; 'x := 2 ; while 'y < 'x do 'y := 'y + 1 ; 'z := 'z + 'y od`
   `initial ; 'x := 2 ; 'y := 'y + 1 ; 'z := 'z + 'y ; 'y := 'y + 1 ; 'z := 'z + 'y`

   (c) `initial ; while 'y < 2 do 'y := 'y + 1 ; 'z := 'z + 'y od [[ 'z ]]`
   `3`

2. Prove that the following program swaps the values of `'x` and `'y`.

   ```
   'x := 'x + 'y ; 'y := 'x - 'y ; 'x := 'x - 'y
   ```

   An executable Maude proof is:

   ```
   th SWAP-PROOF is

     including SEMANTICS .

     op s : -> Store .

   endth

   *** should be: s[['y]]
   reduce s ; 'x := 'x + 'y ; 'y := 'x - 'y ; 'x := 'x - 'y [[ 'x ]] .

   *** should be: s[['x]]
   reduce s ; 'x := 'x + 'y ; 'y := 'x - 'y ; 'x := 'x - 'y [[ 'y ]] .
   ```

3. Write a program that sets `'a` to the maximum of the values of `'x` and `'y`.
   Prove that your program is correct.

   ```
   th  MAX-PROOF  is

     including  SEMANTICS .

     ops  pre post  : Store Int Int -> Bool .

     var  S : Store .
     vars X Y : Int .

     *** some properties of max
     cq  max(X,Y)  =  X  if  Y <= X .
     cq  max(X,Y)  =  Y  if  X <= Y .
   ```

```
  eq  pre(S,X,Y)  =  (S[['x]]) is X and (S[['y]]) is Y .
  eq  post(S,X,Y) =  (S[['a]]) is max(X,Y) .

  op  s : -> Store .
  ops x y : -> Int .

  *** assume precondition
  eq  s[['x]]  =  x .
  eq  s[['y]]  =  y .

endth

*** case analysis: x <= y or y < x

th  CASE1  is

  including  MAX-PROOF .

  eq  x <= y  =  true .

endth

reduce  post(s ; if 'x <= 'y then 'a := 'y else 'a := 'x endif, x, y) .


th  CASE2  is

  including  MAX-PROOF .

  eq  x <= y  =  false .
  *** therefore:
  eq  y < x  =  true .

endth

reduce  post(s ; if 'x <= 'y then 'a := 'y else 'a := 'x endif, x, y) .
```

4. Extend the Maude syntax and semantics of SIMPLE with case-conditionals.

```
fmod CASE is

  extending PROGRAMS .

  sorts Case Cases .
  subsort Case < Cases .

  op _:_ : Numeral Program -> Case [prec 60] .
  op _;;_ : Case Cases -> Cases [ prec 70] .

  op case _ of _ endcase : Expression Cases -> Program .

endfm
```

```
th CASE-SEMANTICS is

  protecting CASE .
  including SEMANTICS .

  op _[[_::_]] : Store Int Cases -> Store .

  var  I : Int .
  var  S : Store .
  var  N : Numeral .
  var  P : Program .
  var  C : Cases .

  cq  S[[ I :: N : P ]]  =  S ; P  if  (S[[ N ]]) == I .
  cq  S[[ I :: N : P ]]  =  S      if  (S[[ N ]]) =/= I .

  cq  S[[ I :: N : P ;; C ]]  =  S ; P          if  (S[[ N ]]) == I .
  cq  S[[ I :: N : P ;; C ]]  =  S[[ I :: C ]]  if  (S[[ N ]]) =/= I .

endth
```

5. (Tricky!) Extend the Maude syntax and semantics of SIMPLE with post-increments: expressions of the form `V++`, which as expressions give the value of the variable `V`, but also have the side-effect of incrementing that value. Comparing this with the Class Test, you will want to make use of pairs `< I , S >`, where `I` is an integer, and `S` is a Store. Such pairs can be specified in Maude as follows.

```
sort  IntStorePair .
op  <_,_>  : Int Store -> IntStorePair .
op  getInt   : IntStorePair -> Int .
op  getStore   : IntStorePair -> Store .
var  I : Int .
var  S : Store .
eq  getInt(< I , S >)  =  I .
eq  getStore(< I , S >)  =  S .
```

You need to change quite a bit. In the theory of Stores, we still need 'table look-up':

```
op  _[[_]]v  : Store Variable -> Int .
```

And then define

```
op  _[[_]]  : Store Expression -> IntStorePair .
```

inductively:

```
var  S : Store .
vars V V' : Variable .
vars E E' : Expression .
var  I : Numeral .

  *** evaluate binary operations, by evaluating their operands
  *** and combining the results (by addition, multiplication, etc.)
eq  S [[ E + E' ]]  =  < getInt(S [[ E ]])
                          + getInt(getStore(S [[ E ]])[[ E' ]]) ,
                         getStore(getStore(S [[ E ]])[[ E' ]]) > .
eq  S [[ E * E' ]]  =  < getInt(S [[ E ]])
                          * getInt(getStore(S [[ E ]])[[ E' ]]) ,
                         getStore(getStore(S [[ E ]])[[ E' ]]) > .

  *** evaluate unary minus by evaluating the operand,
  *** then taking the minus
eq  S [[ - E ]]  =  < - getInt(S[[ E ]]) , getStore(S[[ E ]]) > .

  *** any integer/numeral evaluates to itself
eq  S [[ I ]]  =  < [[I]] , S > .

  *** side effect:
eq  S [[ V ++ ]]  =  < getInt(S [[ V ]]) , S ; V := V + 1 > .

  *** an assignment updates the value associated with the variable ...
eq  S ; V := E [[ V ]]v  =  getInt(S [[ E ]]) .

  *** ... and only that variable
ceq S ; V := E [[ V' ]]  =  getStore(S [[ E ]]) [[ V' ]]v   if V =/= V' .
```