PAPER CODE NO.	EXAMINER : Grant Malcolm
COMP317	DEPARTMENT : Computer Science Tel. No. 795 4244

UNIVERSITY OF LIVERPOOL

SECOND SEMESTER EXAMINATIONS 2010/11

SEMANTICS OF PROGRAMMING LANGUAGES

TIME ALLOWED : Two and a Half Hours

INSTRUCTIONS TO CANDIDATES

Answer FOUR questions.

If you attempt to answer more questions than the required number of questions (in any section), the marks awarded for the excess questions answered will be discarded (starting with your lowest mark).

UNIVERSITY OF LIVERPOOL

1. The syntax and denotational semantics of a simple programming language are summarised in Appendix A. Suppose we want to generalise the assignments in that language to 'parallel assignments'; i.e., programs of the form

$$x_1 \texttt{ := } e_1 \texttt{ \& } x_2 \texttt{ := } e_2 \texttt{ \& } \cdots \texttt{ \& } x_n \texttt{ := } e_n$$

where each assignment is done in parallel rather than sequentially. This means that all the expressions e_i are evaluated in the same state, and their values stored in the corresponding variables x_i. For example, the program

```
'x := 'y & 'y := 'x
```

has the effect of swapping the values stored in the variables `'x` and `'y`. In case a variable occurs more than once in the list $x_1, \ldots, x_n$, all but the final assignment will be ignored. For example,

```
'x := 1 & 'y := 2 & 'x := 2
```

has the same effect as

```
'y := 2 & 'x := 2
```

(a) Modify the BNF definition of programs from Appendix A to allow parallel assignments. (Parallel assignments should consist of one or more assignments to be executed in parallel; to allow for an arbitrary number of assignments, you will need to introduce a new syntactic category $\langle$**AssignmentList**$\rangle$.) **[8 marks]**

(b) Extend the denotational semantics given in Appendix A to give a semantics for parallel assignments. (Just as it was useful in part (a) to introduce a new syntactic category $\langle$**AssignmentList**$\rangle$, it will be useful here to introduce a new semantics function $[\![\_]\!] : \langle \mathbf{AssignmentList} \rangle \times \mathit{State} \to \mathit{State}$.) **[10 marks]**

(c) Use your answer to part (b) to show that the program

```
'x := 0 & 'x := 1
```

sets `'x` to 1. **[7 marks]**

2. For each of the following, give a general definition, and illustrate the definition with reference to the following Maude specification.

```
fmod  ARITHMETIC  is

  sort  Number .

  op  0 : -> Number .
  op  succ : Number -> Number .
  op  plus : Number Number -> Number .

  vars  M N : Number .

  eq  plus(0, N)  =  N .
  eq  plus(succ(M), N)  =  succ(plus(M, N)) .

endfm
```

(a) Signature **[4 marks]**

(b) Σ-algebra **[4 marks]**

(c) Term algebra. **[4 marks]**

(d) Equational theory. **[4 marks]**

(e) Model of an equational theory. **[4 marks]**

(f) Initial model of an equational theory. **[5 marks]**

3. Many languages, such as C and Java, allow assignments to be both programs and expressions. We could modify the language defined in Appendix B by allowing an assignment such as

```
'y := 1
```

to be both a valid program and a valid expression, so that we can also write assignments such as

```
'x := 2 * ('y := 1)
```

which should have the effect of setting `'y` to 1 and `'x` to 2. In general, an assignment of the form V `:=` E, when viewed as an expression, has the value of the expression E; as a side-effect, it also updates the store by assigning the value of E to the variable V.

(a) What changes would you need to make to the module `BASIC-PGM` in order to allow assignments to be both expressions and programs?

[3 marks]

(b) Because expressions may contain assignments, evaluating an expression may change the state. In order to give a semantics to expressions, we still need an operation

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

that gives the value of a variable in a given Store. However, the operation `_[[_]]` that takes an `Expression` and a `Store` as arguments, as in module `STORE`, should now return a pair consisting of the integer value of the expression, together with the updated state that results from evaluating the expression (i.e., if the expression contains assignments, the state will have new values for the variables that are assigned to). Integer-Store 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 .
```

Now the semantics of expressions with side effects can be specified using an operation

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

Give equations defining this operation inductively, as in the module `STORE`. Note that you will need to use the operation `_[[_]]v` for the case where the second argument to `_[[_]]` is a variable. Note that you will also need an equation for the case where the second argument is an assignment.

[15 marks]

(c) We still need the operation

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

that gives the value of a variable in a given Store. Complete the following equation

```
var  V : Variable .
var  E : Expression .
var  S : Store .
eq  S ; V := E [[ V ]]v  =  ...
```

I.e., say what the value of the variable `V` is after the assignment `V := E`.

[3 marks]

(d) Briefly say what other changes would be needed in modules `STORE` and `SEMANTICS`.

[4 marks]

4. The following program, written in the language specified in Appendix B, sets the variable `'z` to the square of the value of `'n` (provided that this value is at least 0), without using multiplication.

```
'x := 0 ;  'y := 0 ; 'z := '0 ;
while not('x is 'n)
do
  'x := 'x + 1 ;
  'y := 'y + 2 ;
  'z := 'z + 'y + 1
od
```

(a) Simplify the following terms (assuming `s` is a `Store`), where *body* is the program

```
'x := 'x + 1 ; 'y := 'y + 2 ; 'z := 'z + 'y + 1 .
```

i. `s ;` *body* `[[ 'x ]]` **[1 mark]**

ii. `s ;` *body* `[[ 'y ]]` **[2 marks]**

iii. `s ;` *body* `[[ 'z ]]` **[3 marks]**

(b) In general, what is meant by an 'invariant' of a loop? **[3 marks]**

(c) Consider the predicate `invy`:

```
op  invy  : Store -> Bool .
var S : Store .
eq  invy(S)  =  (S[[ 'y ]]) == 2 * (S[[ 'x ]]) .
```

Say why `invy` is an invariant of the loop in the program above. **[3 marks]**

(d) Give a pre- and post-condition that specify that `'z` should be set to the square of the value of `'n` (provided that this value is at least 0) **[4 marks]**

(e) Give an invariant that would allow you to prove that the program above is correct with respect to your answer to part (d). **[4 marks]**

(f) Sketch how you would use Maude to prove that the program is correct. **[5 marks]**

5. Linked lists of integers are a data structure used to store sequences of integers. A linked list is either empty ('`null`') or contains both an integer value (the '`head`' of the list) and another linked list (the '`tail`' of the list). An abstract data type of linked lists can be specified in Maude as follows:

```
fmod LINKED_LIST is pr INT.

  sort LList .
  op null : -> LList .
  op head : LList -> Int .
  op tail : LList -> LList .
  op add : Int LList -> LList .

  var L : LList .
  var I : Int .

  eq  head(add(I,L))  =  I .
  eq  tail(add(I,L))  =  L .
endfm
```

Suppose we wanted to extend the language described in Appendix B with a data type of linked lists, so that we could write programs like the following, that computes the sum of all the values in a linked list `list`:

```
'count := 0;
while not isNull(list)
do
  'count := 'count + head(list);
  list := tail(list)
od
```

Here, `list` is a linked-list variable. We can add this to the language of Appendix B by adding the following declarations:

```
sort LLVar .
op list : -> LLVar .
```

We would also need a sort for the linked-list expressions that could be assigned to this variable.

(a) Give further Maude declarations of sorts (e.g., linked-list expressions) and operations (`head`, `tail`, `isNull` and `_:=_`) that will allow programs like the one above in the language. **[10 marks]**

(b) Give Maude equations that describe the semantics of these new constructs, using a new operation `_[[_]]` that takes a `Store` and a linked-list expression, and returns a `LList`. **[15 marks]**

UNIVERSITY OF LIVERPOOL

Appendix A: The Language and its Semantics

Syntax

⟨Exp⟩ ::= ⟨Num⟩ | ⟨Var⟩ | ⟨Exp⟩ + ⟨Exp⟩ | ⟨Exp⟩ - ⟨Exp⟩ | ⟨Exp⟩ * ⟨Exp⟩

⟨BExp⟩ ::= true | false | ⟨Exp⟩ is ⟨Exp⟩ | ⟨Exp⟩ < ⟨Exp⟩
| ⟨BExp⟩ and ⟨BExp⟩ | ⟨BExp⟩ or ⟨BExp⟩ | not ⟨BExp⟩

⟨Pgm⟩ ::= skip | ⟨Var⟩ := ⟨Exp⟩ | ⟨Pgm⟩ ; ⟨Pgm⟩
| if ⟨BExp⟩ then ⟨Pgm⟩ else ⟨Pgm⟩ fi
| while ⟨BExp⟩ do ⟨Pgm⟩ od

Summary of the Denotational Semantics

- $[\![N]\!]_{\text{Exp}}(S) = N$
- $[\![V]\!]_{\text{Exp}}(S) = S(V)$
- $[\![E_1 \texttt{ + } E_2]\!]_{\text{Exp}}(S) = [\![E_1]\!]_{\text{Exp}}(S) + [\![E_2]\!]_{\text{Exp}}(S)$
- $[\![E_1 \texttt{ - } E_2]\!]_{\text{Exp}}(S) = [\![E_1]\!]_{\text{Exp}}(S) - [\![E_2]\!]_{\text{Exp}}(S)$
- $[\![E_1 \texttt{ * } E_2]\!]_{\text{Exp}}(S) = [\![E_1]\!]_{\text{Exp}}(S) * [\![E_2]\!]_{\text{Exp}}(S)$
- $[\![\texttt{true}]\!]_{\text{BExp}}(S) = \mathit{true}$
- $[\![\texttt{false}]\!]_{\text{BExp}}(S) = \mathit{false}$
- $[\![E_1 \texttt{ is } E_2]\!]_{\text{BExp}}(S) = v$, where $v = \mathit{true}$ if $[\![E_1]\!]_{\text{Exp}}(S) = [\![E_2]\!]_{\text{Exp}}(S)$, and $v = \mathit{false}$ otherwise
- $[\![E_1 \texttt{ < } E_2]\!]_{\text{BExp}}(S) = v$, where $v = \mathit{true}$ if $[\![E_1]\!]_{\text{Exp}}(S) < [\![E_2]\!]_{\text{Exp}}(S)$, and $v = \mathit{false}$ otherwise
- $[\![\texttt{not } T]\!]_{\text{BExp}}(S) = \neg\, [\![T]\!]_{\text{BExp}}(S)$
- $[\![T_1 \texttt{ and } T_2]\!]_{\text{BExp}}(S) = [\![T_1]\!]_{\text{BExp}}(S) \wedge [\![T_2]\!]_{\text{BExp}}(S)$
- $[\![T_1 \texttt{ or } T_2]\!]_{\text{BExp}}(S) = [\![T_1]\!]_{\text{BExp}}(S) \vee [\![T_2]\!]_{\text{BExp}}(S)$
- $[\![\texttt{skip}]\!]_{\text{Pgm}}(S) = S$
- $[\![X \texttt{ := } E]\!]_{\text{Pgm}}(S) = S[X \setminus [\![E]\!]_{\text{Exp}}(S)]$
- $[\![P_1 \texttt{ ; } P_2]\!]_{\text{Pgm}}(S) = [\![P_2]\!]_{\text{Pgm}}([\![P_1]\!]_{\text{Pgm}}(S))$
- If $[\![T]\!]_{\text{BExp}}(S) = \mathit{true}$ then $[\![\texttt{if } T \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ fi}]\!]_{\text{Pgm}} = [\![P_1]\!]_{\text{Pgm}}(S)$
- If $[\![T]\!]_{\text{BExp}}(S) = \mathit{false}$ then $[\![\texttt{if } T \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ fi}]\!]_{\text{Pgm}} = [\![P_2]\!]_{\text{Pgm}}(S)$
- If $[\![T]\!]_{\text{BExp}}(S) = \mathit{false}$ then $[\![\texttt{while } T \texttt{ do } P \texttt{ od}]\!]_{\text{Pgm}}(S) = S$
- If $[\![T]\!]_{\text{BExp}}(S) = \mathit{true}$ then $[\![\texttt{while } T \texttt{ do } P \texttt{ od}]\!]_{\text{Pgm}}(S) = [\![\texttt{while } T \texttt{ do } P \texttt{ od}]\!]_{\text{Pgm}}([\![P]\!]_{\text{Pgm}}(S))$

UNIVERSITY OF LIVERPOOL

Appendix B: Maude Semantics

```
*** the programming language: expressions ***
fmod EXPRESSION is pr INT .
                   pr QID *(sort Id to Variable) .
  sort  Expression.
  subsorts  Variable Int < Expression .
  op  _+_  : Expression Expression -> Expression .
  op  _*_  : Expression Expression -> Expression .
  op   -_  : Expression -> Expression .
  op  _-_  : Expression Expression -> Expression .
endfm

fmod BOOL-EXPRESSION is pr EXPRESSION .
  sort BooleanExpression .
  subsort  Bool < BooleanExpression .
  op  _<_ : Expression Expression -> BooleanExpression .
  op  _<=_ : Expression Expression -> BooleanExpression .
  op  _is_ : Expression Expression -> BooleanExpression .
  op  not_ : BooleanExpression -> BooleanExpression .
  op  _and_ : BooleanExpression BooleanExpression -> BooleanExpression .
  op  _or_ : BooleanExpression BooleanExpression -> BooleanExpression .
endfm

*** the programming language: basic programs ***
fmod BASIC-PGM is pr BOOL-EXPRESSION .
  sort  BasicProgram .
  op  _:=_  : Variable Expression -> BasicProgram .
endfm
```

```
*** semantics of basic programs ***
th STORE is pr BASIC-PGM .
  sort Store .

  op  _[[_]] : Store Expression -> Int .
  op  _[[_]] : Store BooleanExpression -> Bool .
  op     _;_ : Store BasicProgram -> Store .
  var  S : Store .
  vars X1 X2 : Variable .
  var  I : Int .
  vars E1 E2 : Expression .
  vars T1 T2 : BooleanExpression .
  var  B : Bool .

  eq  S [[I]]  =  I .
  eq  S [[- E1]]  =  -(S[[E1]]) .
  eq  S [[E1 - E2]]  =  (S[[E1]]) - (S[[E2]]) .
  eq  S [[E1 + E2]]  =  (S[[E1]]) + (S[[E2]]) .
  eq  S [[E1 * E2]]  =  (S[[E1]]) * (S[[E2]]) .

  eq  S [[B]]  =  B .
  eq  S [[E1 is E2]]  =  (S [[E1]]) is (S [[E2]]) .
  eq  S [[E1 <= E2]]  =  (S [[E1]]) <= (S [[E2]]) .
  eq  S [[E1 < E2]]  =  (S [[E1]]) < (S [[E2]]) .
  eq  S [[not T1]]  =  not(S [[T1]]) .
  eq  S [[T1 and T2]]  =  (S [[T1]]) and (S [[T2]]) .
  eq  S [[T1 or T2]]  =  (S [[T1]]) or (S [[T2]]) .

  eq  S ; X1 := E1 [[X1]]  =  S [[E1]] .
  cq  S ; X1 := E1 [[X2]]  =  S [[X2]]   if  X1 =/= X2 .
endth


*** extended programming language ***
fmod PROGRAM is pr BASIC-PGM .
  sort  Program .
  subsort  BasicProgram < Program .
  op  skip  : -> Program .
  op  _;_   : Program Program -> Program .
  op  if_then_else_fi : BooleanExpression Program Program -> Program .
  op  while_do_od     : BooleanExpression Program -> Program .
endfm
```

```
th SEMANTICS is pr PROGRAM .
                pr STORE .
  sort EStore .
  subsort Store < EStore .
  op  _;_ : EStore Program -> EStore .
  var S : Store .
  var T : BooleanExpression .
  var P1 P2 : Program .
  eq  S ; skip  =  S .
  eq  S ; (P1 ; P2)  =  (S ; P1) ; P2 .
  cq  S ; if T then P1 else P2 fi  =  S ; P1
    if  S[[T]] .
  cq  S ; if T then P1 else P2 fi  =  S ; P2
   if  not(S[[T]]) .
  cq  S ; while T do P1 od  =  (S ; P1) ; while T do P1 od
    if  S[[T]] .
  cq  S ; while T do P1 od  =  S
    if  not(S[[T]]) .
endth
```