PAPER CODE NO.	EXAMINER : Grant Malcolm
COMP317	DEPARTMENT : Computer Science Tel. No. 795 4270

UNIVERSITY OF LIVERPOOL

Second Semester Examinations 2014/15

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).

1. Many languages, such as C and Java, allow assignments to be both programs and expressions. We could modify the language defined in Appendix A 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) Modify the BNF syntax of the programming language given in Appendix A so that assignments can be both expressions and programs. (When they are programs, they should be terminated by a semicolon.) **[3 marks]**

(b) In order to give a denotational semantics for expressions with side-effects, we need to change the type of the denotation function $[\![E]\!]_{\mathrm{Exp}}$ for expressions E, so that it returns both the value of the expression and the updated state. I.e., we want to define a denotation function

$$[\![E]\!]_{\mathrm{Exp}} : \mathit{State} \to \mathit{Int} \times \mathit{State}$$

by induction on the form of expressions E. For example, in the case E has the form E_1+E_2, we define:

$$\begin{aligned}&[\![E_1 \texttt{ + } E_2]\!]_{\mathrm{Exp}}(S) = (n_1 + n_2, S_2)\\ &\text{where } (n_1, S_1) = [\![E_1]\!]_{\mathrm{Exp}}(S)\\ &\text{and } (n_2, S_2) = [\![E_2]\!]_{\mathrm{Exp}}(S_1)\ .\end{aligned}$$

This says first evaluate the leftmost expression E_1, giving the integer value n_1 and updated state S_1, then evaluate E_2 in that updated state, giving integer value n_2 and updated state S_2; the value of the expression is $n_1 + n_2$, and evaluation has the side effect of updating the state to S_2.

Complete the inductive definition of $[\![E]\!]_{\mathrm{Exp}}$, i.e., define the function for all possible forms of E, including the case where expression E is an assignment. **[5 marks]**

(c) Use your solution to part (b) to calculate $[\![\texttt{'x := 'x + 1}]\!]_{\mathrm{Exp}}(S)$ for an arbitrary state S. **[3 marks]**

(d) Boolean expressions may contain expressions, so evaluating Boolean expressions may cause the state to change as a side effect. We therefore need to change the type of the denotation function $[\![B]\!]_{\mathrm{BExp}}$ for Boolean expressions B, so that it returns both the value of the Boolean expression and the updated state:

$$[\![B]\!]_{\mathrm{BExp}} : \mathit{State} \to \{\mathit{true}, \mathit{false}\} \times \mathit{State}$$

For example, in the case B has the form E_1<E_2, we define:

$$\begin{aligned}&[\![E_1 \texttt{<} E_2]\!]_{\text{BExp}}(S) = (n_1 < n_2, S_2)\\ &\text{where } (n_1, S_1) = [\![E_1]\!]_{\text{Exp}}(S)\\ &\text{and } (n_2, S_2) = [\![E_2]\!]_{\text{Exp}}(S_1)\ .\end{aligned}$$

(By $n_1 < n_2$, we mean $true$ of $false$, depending on whether or not n_1 is less than n_2.) Complete the inductive definition of $[\![B]\!]_{\text{BExp}}$. **[5 marks]**

(e) The type of the denotation function for programs does not need to be changed; i.e., we still have

$$[\![P]\!]_{\text{Pgm}} : State \to State$$

for all programs P. However, its definition does need to be changed in order to take account of side effects in expressions and Boolean expressions.

i. Modify the definition of $[\![V \texttt{ := } E\texttt{;}]\!]_{\text{Pgm}}$ to take account of the changes in the definition of $[\![E]\!]_{\text{Exp}}$. **[2 marks]**

ii. Complete the inductive definition of $[\![P]\!]_{\text{Pgm}}$. **[4 marks]**

(f) Use your solutions to parts (c) and (e) to show that `'x := 'x := 'x + 1;` is semantically equivalent to `'x := 'x + 1;`. **[3 marks]**

2. Consider 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) Give a general definition of *signature*. **[2 marks]**

(b) Say what the signature of `ARITHMETIC` is. **[2 marks]**

(c) For a signature $\Sigma = (S, O)$, give a general definition of Σ-*terms*. **[2 marks]**

(d) For a signature $\Sigma = (S, O)$, give a general definition of a Σ-*model*. **[3 marks]**

(e) For a signature $\Sigma = (S, O)$, give a general definition of the Σ-*term algebra*, T_Σ. **[3 marks]**

(f) Give two examples of Σ-models (other than the term algebra) where Σ is the signature of `ARITHMETIC`. **[4 marks]**

(g) For a signature Σ and any Σ-model A, there is a homomorphism $h : T_\Sigma \to A$. Give the definition of this homomorphism. **[2 marks]**

(h) For each of the two models in your answer to part (f), give the result of applying the homomorphism from part (g) to the term `plus(succ(0),succ(0))`. **[2 marks]**

(i) For an arbitrary signature Σ, give a general definition of Σ-*equation*. **[2 marks]**

(j) Say what it means for a Σ-model to *satisfy* a Σ-equation. **[3 marks]**

3. **(a)** Describe in detail the process of term-rewriting, and say how a set E of Σ-equations defines a relation $\overset{*}{\leftrightarrow}_E$ on Σ-terms, where $t \overset{*}{\leftrightarrow}_E t'$ if and only if t can be rewritten to t' using the equations in E. **[10 marks]**

(b) Illustrate your answer to part (a) by describing how the term `plus(succ(succ(0)),succ(0))` can be rewritten using the equations in the module `ARITHMETIC` from Question 2 **[4 marks]**

(c) Describe in detail how the relation $\overset{*}{\leftrightarrow}_E$ allows the construction of a (Σ, E)-model, the quotient term algebra, T_Σ/E. **[6 marks]**

(d) What is meant by an *initial* (Σ, E)*-model*? **[2 marks]**

(e) Briefly say why T_Σ/E is an initial (Σ, E)-model. **[3 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 (! 'x == 'n) {
  'z := 'z + 'y + 1 ;
  'y := 'y + 2 ;
  'x := 'x + 1 ;
}
```

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

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

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

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

iii. `[[` *body* `]](s)('z)` **[2 marks]**

(b) In general, what is meant by an ‘invariant’ of a loop? **[4 marks]**

(c) Consider the predicate `invy`:

```
op  invy  : State -> Bool .
var S : State .
eq  invy(S)  =  S('y) is 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. A *hex-counter* is an abstract machine that works with numbers in the range 0..15 in binary notation. It uses an array — let's call it `data` — of length 5 to store binary digits ('0' and '1') with the least significant bit stored at index 0 and the most significant bit stored at index 3; `data`[4] stores a 'carry bit', which is 0 by default, and is set to 1 to indicate that the number stored in the first four components of the array has become larger than 15. For example, to store the number three ('11' in binary), a hex-counter would have

$$\begin{aligned}\texttt{data}[0] &= 1\\ \texttt{data}[1] &= 1\\ \texttt{data}[2] &= 0\\ \texttt{data}[3] &= 0\\ \texttt{data}[4] &= 0\end{aligned}$$

Hex-counters have some basic operations that change the values in the array at a particular index. A hex-counter has a variable — let's call it `index` — that stores the value of the index that is currently in focus; i.e., `index` stores a number in the range 0..4. The *state* of a hex-counter is therefore a pair (I, D), where I is a number in the range 0..4, and D is an array of binary digits of length 5.

The basic commands of hex-counters are:

`inc` which increases the value of `index` by 1 modulo 5, i.e., it sets it to 0 if the value of `index` is 4;

`zero` which sets `data[index]` to 0; and

`one` which sets `data[index]` to 1.

There are also three basic tests:

`index<4` which returns true if the current value of `index` is less than 4; false otherwise;

`zero?` which returns true if the value of `index` is less than 4, *and* `data[index]` is 0; false otherwise; and

`one?` which returns true if the value of `index` is less than 4, *and* `data[index]` is 1; false otherwise.

Hex-counter programs are sequences of basic commands, and also include loops of the form `while (`T`)` $\{\ P\ \}$, where T is a basic test, and P is a program. For example, here's a program that sets `index` to 0, whatever value `index` currently has:

```
while(index<4) {
   inc
}
inc
```

and here's a program that adds 1 to the value stored in `data`, and sets the carry bit to 1 if the number stored was 15, assuming it starts in a state where `index` is 0:

```
while (one?) {
   zero
   inc
}
one
```

(a) Give a BNF definition of the basic hex-counter tests. **[3 marks]**

(b) Give a BNF definition of the hex-counter programs. **[4 marks]**

(c) Define the semantics of basic hex-counter tests; i.e., give an inductive definition of a denotation function $[\![T]\!]_{\text{Test}}$ for any test T, that takes a hex-counter state as argument (i.e., a pair (I, D)), and returns either *true* or *false*. **[5 marks]**

(d) Define the semantics of hex-counter programs; i.e., give an inductive definition of a denotation function $[\![P]\!]_{\text{Pgm}}$ for any program P, that takes a hex-counter state as argument and returns the updated state. For example, given a state (I, D), $[\![\texttt{one}]\!]_{\text{Pgm}}$ should return the state $(I, D[I \leftarrow 1]))$, where '$D[I \leftarrow 1]$' is the array D but with the Ith component set to 1. **[6 marks]**

(e) Write a hex-counter program that, when run in a state where `index` is 0 and `data` contains all 0s, stores the number 5 in `data` (in binary notation, of course), then sets `index` to 0. **[3 marks]**

(f) Let $zeros$ be the array containing only 0s; use your semantics to show that

$$[\![P]\!]_{\text{Pgm}}(0, zeros) = (0, zeros[0 \leftarrow 1][2 \leftarrow 1]) \ ,$$

where P is your program from part (e). **[4 marks]**

Appendix A: The Language SImpL and its Semantics

Syntax

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

⟨BExp⟩ ::= true | false | ⟨Exp⟩==⟨Exp⟩ | ⟨Exp⟩<⟨Exp⟩
         | ⟨BExp⟩&&⟨BExp⟩ | ⟨BExp⟩||⟨BExp⟩ | !⟨BExp⟩

⟨Pgm⟩ ::= skip | ⟨Var⟩ := ⟨Exp⟩; | ⟨Pgm⟩ ⟨Pgm⟩
        | if (⟨BExp⟩) { ⟨Pgm⟩ } else { ⟨Pgm⟩ }
        | while (⟨BExp⟩) { ⟨Pgm⟩ }
```

Summary of the Denotational Semantics

- $[\![N]\!]_{\text{Exp}}(S) = N$
- $[\![V]\!]_{\text{Exp}}(S) = S(V)$
- $[\![E_1 \texttt{+} E_2]\!]_{\text{Exp}}(S) = [\![E_1]\!]_{\text{Exp}}(S) + [\![E_2]\!]_{\text{Exp}}(S)$
- $[\![\texttt{-}E]\!]_{\text{Exp}}(S) = -([\![E]\!]_{\text{Exp}}(S))$
- $[\![E_1 \texttt{*} E_2]\!]_{\text{Exp}}(S) = [\![E_1]\!]_{\text{Exp}}(S) \times [\![E_2]\!]_{\text{Exp}}(S)$
- $[\![\texttt{true}]\!]_{\text{BExp}}(S) = \mathit{true}$
- $[\![\texttt{false}]\!]_{\text{BExp}}(S) = \mathit{false}$
- $[\![E_1 \texttt{==} E_2]\!]_{\text{BExp}}(S) = \mathit{true}$ if $[\![E_1]\!]_{\text{Exp}}(S) = [\![E_2]\!]_{\text{Exp}}(S)$; $false$ otherwise
- $[\![E_1 \texttt{<} E_2]\!]_{\text{BExp}}(S) = \mathit{true}$ if $[\![E_1]\!]_{\text{Exp}}(S) < [\![E_2]\!]_{\text{Exp}}(S)$; $false$ otherwise
- $[\![\texttt{!}\ B]\!]_{\text{BExp}}(S) = \neg\,[\![B]\!]_{\text{BExp}}(S)$
- $[\![B_1 \texttt{\&\&} B_2]\!]_{\text{BExp}}(S) = [\![B_1]\!]_{\text{BExp}}(S) \wedge [\![B_2]\!]_{\text{BExp}}(S)$
- $[\![B_1 \texttt{||} B_2]\!]_{\text{BExp}}(S) = [\![B_1]\!]_{\text{BExp}}(S) \vee [\![B_2]\!]_{\text{BExp}}(S)$
- $[\![\texttt{skip}]\!]_{\text{Pgm}}(S) = S$
- $[\![X \texttt{ := } E;]\!]_{\text{Pgm}}(S) = S[X \leftarrow [\![E]\!]_{\text{Exp}}(S)]$
- $[\![P_1\ P_2]\!]_{\text{Pgm}}(S) = [\![P_2]\!]_{\text{Pgm}}([\![P_1]\!]_{\text{Pgm}}(S))$
- If $[\![B]\!]_{\text{BExp}}(S) = \mathit{true}$ then $[\![\texttt{if}\ (B)\ \{\ P_1\ \}\ \texttt{else}\ \{\ P_2\ \}]\!]_{\text{Pgm}} = [\![P_1]\!]_{\text{Pgm}}(S)$
- If $[\![B]\!]_{\text{BExp}}(S) = \mathit{false}$ then $[\![\texttt{if}\ (B)\ \{\ P_1\ \}\ \texttt{else}\ \{\ P_2\ \}]\!]_{\text{Pgm}} = [\![P_2]\!]_{\text{Pgm}}(S)$
- If $[\![B]\!]_{\text{BExp}}(S) = \mathit{false}$ then $[\![\texttt{while}\ (B)\ \{\ P\ \}]\!]_{\text{Pgm}}(S) = S$
- If $[\![B]\!]_{\text{BExp}}(S) = \mathit{true}$ then $[\![\texttt{while}\ (B)\ \{\ P\ \}]\!]_{\text{Pgm}}(S) = [\![\texttt{while}\ (B)\ \{\ P\ \}]\!]_{\text{Pgm}}([\![P]\!]_{\text{Pgm}}(S))$

Appendix B: Maude Semantics of SIMPLE

```
*** the programming language
fmod SIMPL is pr INT .
              pr QID *(sort Id to Variable) .

  sorts  Exp BExp Program .
  subsorts  Variable Int < Exp .

  op  _+_  : Exp Exp -> Exp .
  op  _*_  : Exp Exp -> Exp .
  op   -_  : Exp -> Exp .

  ops  true false : -> BExp .
  op  _<_ : Exp Exp -> BExp .
  op  _==_ : Exp Exp -> BExp .
  op  !_ : BExp -> BExp .
  op  _&&_ : BExp BExp -> BExp .
  op  _||_ : BExp BExp -> BExp .

  op  skip  : -> Program .
  op  _:=_;  : Variable Exp -> Program .
  op  _ _   : Program Program -> Program .
  op  if(_){_}else{_} : BExp Program Program -> Program .
  op  while(_){_}     : BExp Program -> Program .
endfm


*** theory of storage ***
th STATE is pr INT .
            pr QID *(sort Id to Variable) .
  sort State .

  op  _(_) : State Variable -> Int .
  op  _[_<-_] : State Variable Int -> State .

  vars X1 X2 : Variable .
  var  S : State .
  var  I : Int .

  eq  (S[X1 <- I])(X1)  =  I .
  cq  (S[X1 <- I])(X2)  =  S(X2)   if  X1 =/= X2 .
endth
```

```
th SEMANTICS is pr SIMPL .
                inc STATE .
  op  [[_]](_) : Exp State -> Int .
  op  [[_]](_) : BExp State -> Bool .

  var S : State .
  var P1 P2 : Program .
  var  I : Int .
  vars E1 E2 : Exp .
  vars T T1 T2 : BExp .
  var  B : Bool .
  var  V : Variable .

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

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

  sort ErrorState .
  subsort State < ErrorState .
  op  [[_]](_) : Program ErrorState -> ErrorState .

  eq  [[skip]](S)  =  S .
  eq  [[ V := E1 ;]](S)  =  S[ V <- [[E1]](S) ] .
  eq  [[P1  P2]](S)  =  [[P2]]([[P1]](S)) .
  cq  [[if (T) { P1 } else { P2 }]](S)  =  [[P1]](S)
    if [[T]](S) .
  cq  [[if (T) { P1 } else { P2 }]](S)  =  [[P2]](S)
    if not([[T]](S)) .
  cq  [[while (T) { P1 }]](S)  =  [[while (T) { P1 }]]([[P]](S))
    if  S[[T]] .
  cq  [[while (T) { P1 }]](S)  =  S
    if  not(S[[T]]) .
endth
```