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

UNIVERSITY OF LIVERPOOL

Second Semester Examinations 2011/12

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. Languages such as Java allow the programmer to write methods that consist of a series of program instructions and, if the return type is non-void, end with a `return` command. Calls of such methods can then be used in expressions. Without adding method-calls to the syntax of the simple language defined in Appendix A, we can add 'method bodies' by extending the syntax of the language to include Expressions of the form

$$\texttt{do}\ \{\ P\ \texttt{return}\ E\ \}$$

where P is a program and E is an expression. Such expressions are evaluated by executing the program P (which will have the side-effect of updating the state), then evaluating the expression E: the value obtained from E is the value of the whole expression. For example, when the expression `do {  'x := 1 ; 'y := 'x + 1  return 'x + 'y }` is evaluated, it will return the value 3, and it will have the side-effect of setting `'x` to 1 and `'y` to 2.

(a) Modify the BNF definition of Expressions from Appendix A to allow for 'method bodies' as described above. **[4 marks]**

(b) In order to give a denotational semantics for expressions with side-effects, we need to change the type of the denotation function $[\![E]\!]_{\text{Exp}}$ 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

$$[\![E]\!]_{\text{Exp}} : \mathit{Store} \to \mathit{Int} \times \mathit{Store}$$

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

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

This says first evaluate the 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.

i. Complete the inductive definition of $[\![E]\!]_{\text{Exp}}$, including the case where expression E is a 'method body'. **[10 marks]**

ii. Modify the definition of $[\![V \texttt{ := } E]\!]_{\text{Pgm}}$ to take account of the changes in the definition of $[\![E]\!]_{\text{Exp}}$. **[4 marks]**

iii. What other changes would need to be made to the denotation of Boolean Expressions? **[2 marks]**

iv. Use your answers to parts (i) and (ii) to calculate the semantics of the following assignment.

```
'x := do { 'x := 'x + 1 return 2 * 'x }
```

[5 marks]

2. For each of the following, give a detailed 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-model. **[4 marks]**

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

(d) Σ-equation **[4 marks]**

(e) Satisfaction of a Σ-equation by a Σ-model. **[4 marks]**

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

3. What exactly is an invariant of a loop? Describe how invariants can be used to prove the correctness of loops, and say why it is a valid form of reasoning. **[25 marks]**

4. The following program, written in the language specified in Appendix B, computes exponentials. Specifically, it sets the variable `'e` to the value of $'x^{'y}$:

```
'e := 1 ;  'i := 0;
while not('i is 'y)
do
  'e := 'x * 'e ;
  'i = 'i + 1
od
```

(a) Assuming `'x` has the value 3, and `'y` has the value 2, how often is the body of the while-loop executed? Give the values of `'e` and `'i` after each iteration of the body of the loop. **[4 marks]**

(b) Give a suitable precondition and postcondition to specify that the program sets `'e` to the value of `'x` raised to the power of the value of `'y`. (Maude notation for exponentiation is _^_.) **[5 marks]**

(c) Give a suitable invariant for the loop, which will allow you to prove the correctness of the program. **[8 marks]**

(d) Give a Maude proof score that will prove correctness of the program. **[8 marks]**

5. A *2-register machine* is an abstract machine that has two 'registers', `r1` and `r2`, that each store an integer value. The values in these registers are updated by programs, which are sequences of instructions. The basic instructions are as follows.

- `inc1` and `inc2`, which add 1 to the value stored in `r1` and `r2`, respectively; i.e., `inc1` adds 1 to the value stored in `r1` and leaves the value in `r2` unchanged, and similarly for `inc2`.
- `dec1` and `dec2`, which subtract 1 from the value stored in `r1` and `r2`, respectively.
- `copy`, which sets the value of `r2` to the value of `r1`, and sets the value of `r1` to 0.

Programs are sequences of instructions, with a loop construct of the form

`while r2>0` $\{\ P\ \}$

where P is a program. This repeatedly executes P while the value stored in `r2` is greater than 0; if the value stored in `r2` is less than or equal to 0, the program does nothing. For example, the following program, which we'll call `double`, doubles the value stored in `r1`.

```
copy
while r2>0 {
   dec2
   inc1
   inc1
}
```

(a) Give a BNF specification of programs. **[3 marks]**

(b) Give a Maude specification of programs, using a sort `Program`, so that 2-register programs are terms of sort `Program`. **[4 marks]**

(c) Briefly say why `double` is a well-formed term of sort `Program`. **[2 marks]**

(d) Give a Maude specification of the semantics of programs. (*Hint*: the state of a 2-register machine is just a pair of integers; specify pairs of integers and give equations describing the effects of programs on these pairs.) **[10 marks]**

(e) Use your answer to part (d) to show that, when `double` is run in a state where the value in `r1` is 2, it results in a state where `r1` stores the value 4. **[6 marks]**

Appendix A: The Language and its Semantics

Syntax

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

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

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

Summary of the Denotational Semantics

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

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 TST 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 BPGM is pr TST .
  sort  BasicProgram .
  op  _:=_  : Variable Expression -> BasicProgram .
endfm
```

```
*** semantics of basic programs ***
th STORE is pr BPGM .
  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 PGM is pr BPGM .
  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 SEM is pr PGM .
          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
```