



UNIVERSITY OF
LIVERPOOL

MAY EXAMINATIONS 2010

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. Appendix A summarises the syntax and denotational semantics of a simple imperative programming language. This question asks you to extend the syntax and semantics of that language with two new constructs.

- (a) Many languages, such as C and Java, have a ‘ternary conditional’: an operator that takes one boolean and two other arguments and returns the first of these if the boolean argument is true, and returns the second if the boolean argument is false. We would like to introduce such an operator into our language. For example, we would like expressions such as

$$('x < 0) ? 1 - 'x : 2 * 'x$$

or, in general, expressions of the form

$$(T) ? E_1 : E_2$$

where T is a boolean expression (a ‘Tst’), and E_1 and E_2 are expressions (‘Exps’). Such expressions are evaluated as follows: first, the boolean expression T is evaluated. If this is true, then the result of the entire expression is the value of the expression E_1 ; if T evaluates to false, then the result is the value of E_2 .

- i. Extend the syntax of the language in Appendix A with ternary conditionals. **[2 marks]**
- ii. Extend the semantics of the language in Appendix A by describing formally how ternary conditionals are evaluated. **[3 marks]**
- iii. Use your semantics to show that the assignment

$$'x := ('y < 0) ? 0 : 1$$

has the same semantics as the program

$$\text{if } 'y < 0 \text{ then } 'x := 0 \text{ else } 'x := 1 \text{ fi .}$$

[5 marks]

- (b) ‘Multiple-if’ statements are programs of the form

```
multi-if
  T1 : P1 ;;
  T2 : P2 ;;
  ...   ;;
  Tm : Pm
end-if
```

where each T_i is a boolean expression and each P_i is a program. A multi-if statement consists, therefore, of a list of *clauses* of the form $T_i : P_i$, separated by ‘;;’ and contained inside the delimiters `multi-if` and `end-if`. In a given state S , such a program is executed as follows: first, all the boolean expressions T_i ($0 < i \leq m$) are evaluated in the state S (i.e., all tests are evaluated in the *same* state); then *each* program P_i is executed only if the corresponding test T_i evaluated to true. Note that all the programs whose corresponding tests evaluate to true will be executed in the order in which they are listed. For example, the program

```

'x := 0 ;
multi-if
  'x >= 0 : 'x := 'x + 1 ;;
  'x is 0  : 'x := 'x * 2 ;;
  'x < 0   : 'x := 0
end-if

```

will end up setting the value of 'x to 4, because both the first and the second clauses will evaluate to true after the initial assignment ('x := 0), so first the program 'x := 'x + 1 will be executed, then the program 'x := 'x * 2.

- i. Give a BNF specification of a syntactic category, $\langle \text{ClauseList} \rangle$, of lists of clauses separated by ';', where each clause is of the form $T : P$, where T is a boolean expression and P a program. **[2 marks]**
- ii. Extend the BNF specification of programs given in Appendix A with multi-if statements. **[1 mark]**
- iii. Use your answer to part (i) to define a function

$$\llbracket CL \rrbracket_{\text{ClauseList}} : \text{State} \times \text{State} \rightarrow \text{State}$$

by induction on the $\langle \text{ClauseList} \rangle$ CL so that for all states S and S' ,

$$\llbracket CL \rrbracket_{\text{ClauseList}}(S, S')$$

is the state that results from executing, in S' , all the programs in all the clauses in CL whose corresponding tests evaluate to true in state S . **[5 marks]**

- iv. Use the function defined in part (iii) to extend the semantics of Appendix A to multiple-if statements by completing the following: for any ClauseList CL ,

$$\llbracket \text{multi-if } CL \text{ end-if} \rrbracket_{\text{Pgm}}(S) = \dots$$

[2 marks]

- v. Use your answers to parts (iii) and (iv) to show that for any boolean expression T and programs P_1 and P_2 , the program

```
multi-if T : P1 ;; not(T) : P2 endif
```

has the same semantics as

```
if T then P1 else P2 fi .
```

[5 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. Describe term rewriting in detail, and illustrate the process by describing how Maude would reduce the term

`plus(succ(succ(0)), succ(0))`

given the ARITHMETIC specification listed in Question 2. [25 marks]

4. The following program, written in the language specified in Appendix B, computes powers of 2. Specifically, it sets the variable 'x to the value of 2^y :

```
'x := 1 ; 'i := 0 ;
while 'i < 'y
do
  'x := 2 * 'x ;
  'i = 'i + 1
od
```

- (a) Briefly describe what it means for a program to be correct with respect to a given pre- and post-condition, and say why invariants can be used to prove the correctness of programs. **[5 marks]**
- (b) Give a suitable precondition and postcondition to specify that the program sets 'x to the value of 2^y . (Maude notation for exponentiation is `_**_`.) **[4 marks]**
- (c) Give a suitable invariant for the loop, which will allow you to prove the correctness of the program. **[5 marks]**
- (d) Give a proof score that will prove the correctness of the program. **[7 marks]**
- (e) For the same pre- and post-conditions, what invariant would you propose to prove the correctness of the following program?

```
'x := 1 ; 'i := 'y ;
while 'i > 0
do
  'x := 2 * 'x ;
  'i = 'i - 1
od
```

[4 marks]

5. An abstract data type of pairs of integers is given in the following Maude specification:

```
fmod PAIR is
  protecting INT .

  sort Pair .

  op <_,_> : Int Int -> Pair .
  ops (fst_) (snd_) : Pair -> Int .

  vars I J : Int .

  eq  fst < I , J > = I .
  eq  snd < I , J > = J .

endfm
```

We want to extend the programming language described in Appendix B with a data type of pairs, so that we can write programs such as the following:

```
q := < 1 , 2 > ; (p).1 := (q).2 ; (p).2 := (q).1
```

where

- p and q are variables of the programming language, and store pairs of integers;
- $(_).1$ and $(_).2$ refer to the first and second components of a pair;
- $\langle E1, E2 \rangle$ represents a pair whose first component is the value of the integer expression $E1$ and whose second component is the value of the integer expression $E2$; and
- the overloaded operator $_ := _$ allows assignments either to a ‘pair variable’ such as p or q , or to a component of a pair variable, such as $(p).1$ or $(p).2$.

This program sets q to a pair whose first component is 1 and whose second component is 2, then sets the first component of p to the second component of q (i.e., the value 2), and finally sets the second component of p to the first component of q . After the program has run, q has the value $\langle 1, 2 \rangle$ and p has the value $\langle 2, 1 \rangle$.

- (a) Specify the syntax of the extended language by completing the following Maude specification with subsort and operator declarations (one of the overloaded assignment operators has been declared for you).

```
fmod PAIR-PROGRAMS is extending PROGRAM .

  *** Variables of the programming language:
  sort PairVar .
  ops p q : -> PairVar .
```

```
*** First and second components of pairs:
sort PairComponent .

*** Expressions of type Pair:
sort PairExp .

*** Subsort declarations:

*** Operations of the language:
op _:=_ : PairComponent Expression -> BasicProgram .

endfm
```

Complete this specification by adding declarations of operators $(_).1$ and $(_).2$, $\langle _, _ \rangle$, and an assignment operation that allows assignment to pair variables. Also give two subsort declarations that allow pair components such as $(p).1$ to occur on the left- and right-hand sides of assignments, and also allow pair variables to be pair expressions. **[7 marks]**

- (b) The semantics of the extended language can be specified by overloading the operator $_ [[_]]$ as in the following Maude module:

```
th PAIR-SEMANTICS is protecting SEMANTICS .
    protecting PAIR .
    protecting PAIR-PROGRAMS .

op _[[[_]] : Store PairExp -> Pair .

endth
```

Define the semantics of the extended language by giving suitable equations to include in PAIR-SEMANTICS. **[12 marks]**

- (c) Use the equations in your answer to part (b) to simplify the following term:

$(s ; q := \langle 1, 2 \rangle ; (p).1 := (q).2 ; (p).2 := (q).1)[[p]]$

for a given Store s . **[6 marks]**

Appendix A: The Language and its Semantics

Syntax

$$\langle \text{Exp} \rangle ::= \langle \text{Num} \rangle \mid \langle \text{Var} \rangle \mid \langle \text{Exp} \rangle + \langle \text{Exp} \rangle \mid \langle \text{Exp} \rangle - \langle \text{Exp} \rangle \mid \langle \text{Exp} \rangle * \langle \text{Exp} \rangle$$

$$\langle \text{Tst} \rangle ::= \text{true} \mid \text{false} \mid \langle \text{Exp} \rangle \text{ is } \langle \text{Exp} \rangle \mid \langle \text{Exp} \rangle < \langle \text{Exp} \rangle \\ \mid \langle \text{Tst} \rangle \text{ and } \langle \text{Tst} \rangle \mid \langle \text{Tst} \rangle \text{ or } \langle \text{Tst} \rangle \mid \text{not } \langle \text{Tst} \rangle$$

$$\langle \text{Pgm} \rangle ::= \text{skip} \mid \langle \text{Var} \rangle := \langle \text{Exp} \rangle \mid \langle \text{Pgm} \rangle ; \langle \text{Pgm} \rangle \\ \mid \text{if } \langle \text{Tst} \rangle \text{ then } \langle \text{Pgm} \rangle \text{ else } \langle \text{Pgm} \rangle \text{ fi} \\ \mid \text{while } \langle \text{Tst} \rangle \text{ do } \langle \text{Pgm} \rangle \text{ od}$$

Summary of the Denotational Semantics

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

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 \neq 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
```