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

UNIVERSITY OF LIVERPOOL

# Second Semester Examinations 2013/14

# 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 extend the syntax of expressions with a post-increment operator, to include expressions of the form $V$++, where $V$ is a variable (i.e., of syntactic class $\langle \mathbf{Var} \rangle$). In any state $S$, the value of the expression $V$++ is just the value of $V$ in $S$, but evaluating the expression has the side-effect of updating the state so that the value of $V$ is incremented by 1.

**(a)** Give a BNF description of the syntax of expressions that includes expressions of the form $V$++. **[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

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

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

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

This says first evaluate the leftmost expression $E_1$, giving the integer value $n_1$ and updated state $S_1$, then evaluate $E_2$ in that updated state, giving the 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 $E$ is of the form $V$++. **[10 marks]**

ii. Modify the definition of $[\![V \texttt{ := } E]\!]_{\text{Pgm}}$ to take account of the changes in the definition of $[\![E]\!]_{\text{Exp}}$. **[6 marks]**

iii. What other changes would need to be made to the denotational semantics of the language? **[5 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. **[3 marks]**

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

**(d)** Give an explicit definition of $\Sigma$-terms where $\Sigma$ is the signature of `ARITHMETIC`. **[2 marks]**

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

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

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

**(h)** For a signature $\Sigma$ and any $\Sigma$-model $A$, there is a homomorphism $h : T_\Sigma \to A$. Give the definition of this homomorphism. **[3 marks]**

**(i)** For each of the two models in your answer to part (g), give the result of applying the homomorphism from part (h) to the term `plus(succ(0),succ(0))`. **[2 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 integer division of `'x` by `2`, provided that `'x` is at least `0`. Specifically, it sets `'p` to the result of dividing `'x` by `2`, and sets `'q` to the remainder after dividing `'x` by `2`.

```
'q := 'x ;  'p := 0;
while 'q > 1
do
  'p := 'p + 1 ;
  'q := 'q - 2
od
```

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

```
'p := 'p + 1 ; 'q := 'q - 2 .
```

i. `s ;` *body* `[[ 'x ]]` **[2 marks]**

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

iii. `s ;` *body* `[[ 'q ]]` **[2 marks]**

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

**(c)** Consider the predicate `inv1`:

```
op  inv1  : Store -> Bool .
var S : Store .
eq  inv1(S)  =
      (S[[ 'x ]]) == 2 * (S[[ 'p ]]) + (S[[ 'q ]]) .
```

Say why `inv1` is an invariant of the loop in the program above. **[3 marks]**

**(d)** Give a pre- and post-condition that specify that `'p` and `'q` are the result and remainder, respectively, of the integer division of `'x` by `2` (provided that `'x` 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 *3-register machine* is an abstract machine that has three 'registers', `r1`, `r2` and `r3`, that each store an integer value. The values in these registers are initially all 0, and are updated by programs, which are sequences of instructions. The basic instructions are as follows.

- `r2 :=` *N*, where *N* is a number. This sets the value of `r2` to be *N*.
- `dec1`, which subtracts 1 from the value stored in `r1` and leaves the values in `r2` and `r3` unchanged.
- `copy`, which sets the value of `r1` to the value of `r2`, and leaves the values of `r2` and `r3` unchanged.
- `add`, which adds the value stored in `r2` to the value stored in `r3`, and leaves the values stored in `r1` and `r2` unchanged.

There are also two basic *tests*:

- `r1 > 0`, which is true if the value stored in `r1` is greater than 0, and false otherwise.
- `r2 == r3`, which is true if the values stored in `r2` and `r3` are equal, and false otherwise.

In addition, there is a prefix negation operation, `!`, and an infix logical conjunction, `&`. For example, the test

```
r1 > 0 & !(r2 == r3)
```

is true only when the value in `r1` is greater than 0 and the values in `r2` and `r3` are different.

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

`while (`$T$`) {` $P$ `}`

where $T$ is a test and $P$ is a program. This repeatedly executes $P$ while the test $T$ evaluates to true. For example, the following program, which we'll call `addTwice`, sets the value stored in `r3` to 7.

```
r2 := 3
add
r2 := 2
copy
while (r1 > 0) {
   dec1
   add
}
```

The first line sets `r2` to 3; `add` then sets r3 to 3; `r2` is then set to 2 and this value is copied to `r1`; the loop then repeats twice, decrementing `r1` and adding 2 to `r3`.

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

**(b)** Give a BNF specification of programs (you may assume a syntactic category <Num> of numbers is given). **[4 marks]**

**(c)** Specify the semantics of tests. (*Hint*: the state of a 3-register machine is just a 3-tuple of integers; specify a denotation function that takes a test and a 3-tuple of integers as arguments and returns true or false when the test is true in that state.) **[5 marks]**

**(d)** Specify the semantics of programs. (*Hint*: specify a denotation function that takes a program and returns a partial function from states to states; i.e., define $[\![P]\!]_{\text{Pgm}}(S)$ for all programs $P$ and states $S$.) **[8 marks]**

**(e)** Use your answer to part (d) to show that, when `addTwice` is run in a state where the value in each register is 0, it results in a state where `r1` stores the value 0, `r2` stores the value 2, and `r3` stores the value 7. **[5 marks]**

UNIVERSITY OF LIVERPOOL

# Appendix A: The Language SIMPLE and its Semantics

## Syntax

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

⟨BExp⟩ ::= true | false | ⟨Exp⟩ == ⟨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

- $[\![N]\!]_{\text{Exp}}(S) = N$
- $[\![V]\!]_{\text{Exp}}(S) = S(V)$
- $[\![E_1 \texttt{ + } E_2]\!]_{\text{Exp}}(S) = [\![E_1]\!]_{\text{Exp}}(S) + [\![E_2]\!]_{\text{Exp}}(S)$
- $[\![E_1 \texttt{ - } E_2]\!]_{\text{Exp}}(S) = [\![E_1]\!]_{\text{Exp}}(S) - [\![E_2]\!]_{\text{Exp}}(S)$
- $[\![E_1 \texttt{ * } E_2]\!]_{\text{Exp}}(S) = [\![E_1]\!]_{\text{Exp}}(S) * [\![E_2]\!]_{\text{Exp}}(S)$
- $[\![\texttt{true}]\!]_{\text{Tst}}(S) = \mathit{true}$
- $[\![\texttt{false}]\!]_{\text{Tst}}(S) = \mathit{false}$
- $[\![E_1 \texttt{ == } E_2]\!]_{\text{Tst}}(S) = v$, where $v = \mathit{true}$ if $[\![E_1]\!]_{\text{Exp}}(S) = [\![E_2]\!]_{\text{Exp}}(S)$, and $v = \mathit{false}$ otherwise
- $[\![E_1 \texttt{ < } E_2]\!]_{\text{Tst}}(S) = v$, 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 of SIMPLE

```
*** 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 BOOLEAN-EXPRESSION is pr EXPRESSION .
  sort BooleanExpression .
  ops  true false : -> 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 BOOLEAN-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 == E2]]  =  (S [[E1]]) == (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 PROGRAMS 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 SEM is pr PROGRAMS .
          pr STORE .
  sort ErrorStore .
  subsort Store < ErrorStore .
  op  _;_ : ErrorStore Program -> ErrorStore .
  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
```