PAPER CODE NO. **COMP317**	EXAMINER : Grant Malcolm DEPARTMENT : Computer Science	TEL. NO. 794 6794

THE UNIVERSITY *of* LIVERPOOL

JANUARY 2003 EXAMINATIONS

Bachelor of Arts : Year 3
Bachelor of Engineering : Year 3
Bachelor of Science : Year 3

SEMANTICS OF PROGRAMMING LANGUAGES

TIME ALLOWED : Two Hours and a Half

INSTRUCTIONS TO CANDIDATES

Answer **four** questions only.

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

THE UNIVERSITY
of LIVERPOOL

1. Some languages, such as Dijkstra's Guarded Command Language, allow non-deterministic programs; for example, given Boolean expressions b_1 and b_2 and programs c_1 and c_2, the program

$$(b_1 \to c_1) \,[\!]\, (b_2 \to c_2)$$

is evaluated as follows: starting in a state s, the guards b_1 and b_2 are evaluated;

- if neither guard evaluates to true, the program fails ('crashes');
- if exactly one of the guards evaluates to true, then the corresponding program is executed in state s (i.e., if b_1 is true, then c_1 is executed; if b_2 is true, then c_2 is executed);
- if both guards evaluate to true, then either c_1 or c_2 is executed (i.e., a non-deterministic choice is made as to which program is executed).

For example, the following program sets `z` to the minimum of the values of `x` and `y`:

```
(x <= y  →  z := x) [] (y <= x  →  z := y)
```

(a) Extend the syntax of IMP programs to include programs of the form

$$(b_1 \to c_1) \,[\!]\, (b_2 \to c_2)$$

[10 marks]

(b) Extend the operational semantics of IMP to give a semantics for these programs (the syntax and operational semantics of IMP are summarised in Appendix A). **[15 marks]**

2. The syntax of the programming language IMP and the denotational semantics for arithmetic expressions are summarised in Appendix A. Suppose we want to extend the syntax of arithmetic expressions in IMP with a post-increment operator, to include expressions of the form x++, where x is a variable (i.e., of syntactic class $\langle\mathbf{Loc}\rangle$). In any state s, the value of the expression x++ is just the value of x in s, but evaluating the expression has the side-effect of updating the state so that x is incremented by `1`.

   (a) Give a BNF description of the syntax of arithmetic expressions that includes expressions of the form x++. **[10 marks]**

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

   $$\mathcal{A}[\![a]\!] : \mathit{State} \to \mathit{Int} \times \mathit{State}$$

   by induction on the form of arithmetic expressions a. For example, in the case a has the form a_1+a_2, we define:

   $$\mathcal{A}[\![a_1 + a_2]\!](s) = n + \mathcal{A}[\![a_2]\!](s') \text{ where } \mathcal{A}[\![a_1]\!](s) = (n, s') \;.$$

   This says first evaluate the leftmost expression a_1, giving the integer value n and updated state s', then evaluate a_2 in that updated state.

      i. Complete the inductive definition of $\mathcal{A}[\![a]\!]$, including the case where a is of the form x++. **[10 marks]**

      ii. Modify the definition of $\mathcal{C}[\![x\ := a]\!]$ to take account of the changes in the definition of $\mathcal{A}[\![a]\!]$. **[5 marks]**

THE UNIVERSITY
of LIVERPOOL

3. The axiomatic semantics of IMP is summarised in Appendix B. The following program sets the variable x to the value of 2^y:

```
x := 1 ;
while not(y = 0)
do
  x := 2 * x ;
  y := y - 1
```

(a) Give a suitable precondition and postcondition to specify that the program sets x to the value of 2^y. **[5 marks]**

(b) Give a suitable invariant for the loop, which will allow you to prove the correctness of the program with respect to the pre- and post-conditions you gave in part (a). **[10 marks]**

(c) Give a proof of the correctness of the program in the annotated-program style. **[10 marks]**

4. The OBJ semantics of a simple imperative language is given in Appendix C.

(a) Give an OBJ proof script that shows that the following program sets `'Z` to the square of `'X+'Y`:

```
'Z := 'X + 'Y ;
'X := 'Z * 'X ;
'Y := 'Z * 'Y ;
'Z := 'X + 'Y
```

[8 marks]

(b) Do the reduction in your answer to part (a) by hand. **[12 marks]**

(c) What property of multiplication and addition is needed for the proof to work? Express this property as an OBJ equation. **[5 marks]**

5. (a) Give a constructive proof and proof-witness of

$$(A \wedge C \Rightarrow B) \Rightarrow C \Rightarrow A \Rightarrow B\ .$$

[9 marks]

(b) Give a constructive proof and proof-witness of

$$A \wedge (A \Rightarrow B) \Rightarrow B\ .$$

[9 marks]

(c) Replacing C with $A \Rightarrow B$ in the proposition in part (a) allows us to apply the proof-witness from part (a) to the proof-witness from part (b), giving a λ-term that is a proof-witness of

$$(A \Rightarrow B) \Rightarrow A \Rightarrow B\ .$$

Reduce this λ-term as far as possible. **[7 marks]**

THE UNIVERSITY
of LIVERPOOL

Appendix A: Syntax and Semantics of IMP

Syntax of IMP

```
⟨Aexp⟩ ::= ⟨Num⟩ | ⟨Loc⟩ | ⟨Aexp⟩ + ⟨Aexp⟩
           | ⟨Aexp⟩ - ⟨Aexp⟩ | ⟨Aexp⟩ * ⟨Aexp⟩

⟨Bexp⟩ ::= true | false | ⟨Aexp⟩ = ⟨Aexp⟩ | ⟨Aexp⟩ < ⟨Aexp⟩
           | ⟨Bexp⟩ and ⟨Bexp⟩ | ⟨Bexp⟩ or ⟨Bexp⟩ | not ⟨Bexp⟩

⟨Com⟩ ::= skip | ⟨Loc⟩ := ⟨Aexp⟩ | ⟨Com⟩ ; ⟨Com⟩
          | if ⟨Bexp⟩ then ⟨Com⟩ else ⟨Com⟩
          | while ⟨Bexp⟩ do ⟨Com⟩
```

Summary of the Operational Semantics

Operational semantics of programs:

- $(\texttt{skip}, s) \to s'$ if and only if $s = s'$
- If $(a, s) \to n$ then $(x \texttt{ := } a, s) \to s[n/x]$
- If $(c_1, s) \to s'$ and $(c_2, s') \to s''$ then $(c_1 \texttt{ ; } c_2, s) \to s''$
- If $(b, s) \to \mathit{true}$ and $(c_1, s) \to s'$ then $(\texttt{if}\, b\, \texttt{then}\, c_1\, \texttt{else}\, c_2, s) \to s'$
- If $(b, s) \to \mathit{false}$ and $(c_2, s) \to s'$ then $(\texttt{if}\, b\, \texttt{then}\, c_1\, \texttt{else}\, c_2, s) \to s'$
- If $(b, s) \to \mathit{false}$ then $(\texttt{while}\, b\, \texttt{do}\, c, s) \to s$
- If $(b, s) \to \mathit{true}$ and $(c, s) \to s'$ and $(\texttt{while}\, b\, \texttt{do}\, c, s') \to s''$
  then $(\texttt{while}\, b\, \texttt{do}\, c, s) \to s''$

Summary of the Denotational Semantics

- $\mathcal{A}[\![n]\!](s) = n$
- $\mathcal{A}[\![x]\!](s) = s(x)$
- $\mathcal{A}[\![a_1 \texttt{ + } a_2]\!](s) = \mathcal{A}[\![a_1]\!] + \mathcal{A}[\![a_2]\!]$
- $\mathcal{A}[\![a_1 \texttt{ - } a_2]\!](s) = \mathcal{A}[\![a_1]\!] - \mathcal{A}[\![a_2]\!]$
- $\mathcal{A}[\![a_1 \texttt{ * } a_2]\!](s) = \mathcal{A}[\![a_1]\!] * \mathcal{A}[\![a_2]\!]$
- $\mathcal{B}[\![\texttt{true}]\!](s) = \mathit{true}$
- $\mathcal{B}[\![\texttt{false}]\!](s) = \mathit{false}$
- $\mathcal{B}[\![a_1 \texttt{ = } a_2]\!](s) = v$, where $v = \mathit{true}$ if $\mathcal{A}[\![a_1]\!](s) = \mathcal{A}[\![a_2]\!](s)$, and $v = \mathit{false}$ otherwise

- $\mathcal{B}[\![a_1 < a_2]\!](s) = v$, where $v = \mathit{true}$ if $\mathcal{A}[\![a_1]\!](s) < \mathcal{A}[\![a_2]\!](s)$, and $v = \mathit{false}$ otherwise
- $\mathcal{B}[\![\texttt{not}\ b]\!](s) = \neg\, \mathcal{B}[\![b]\!](s)$
- $\mathcal{B}[\![b_1\ \texttt{and}\ b_2]\!](s) = \mathcal{B}[\![b_1]\!](s) \wedge \mathcal{B}[\![b_2]\!](s)$
- $\mathcal{B}[\![b_1\ \texttt{or}\ b_2]\!](s) = \mathcal{B}[\![b_1]\!](s) \vee \mathcal{B}[\![b_2]\!](s)$
- $\mathcal{C}[\![\texttt{skip}]\!](s) = s$
- $\mathcal{C}[\![x\ \texttt{:=}\ a]\!](s) = s[n/x]$
- $\mathcal{C}[\![c_1\ \texttt{;}\ c_2]\!](s) = \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!](s))$
- If $\mathcal{B}[\![b]\!](s) = \mathit{true}$ then $\mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] = \mathcal{C}[\![c_1]\!](s)$
- If $\mathcal{B}[\![b]\!](s) = \mathit{false}$ then $\mathcal{C}[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] = \mathcal{C}[\![c_2]\!](s)$
- If $\mathcal{B}[\![b]\!](s) = \mathit{false}$ then $\mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!] = s$
- If $\mathcal{B}[\![b]\!](s) = \mathit{true}$ then $\mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!] = \mathcal{C}[\![\texttt{while}\ b\ \texttt{do}\ c]\!](\mathcal{C}[\![c]\!](s))$

Appendix B: Hoare Logic

$$\{\,A\,\}\ \texttt{skip}\ \{\,A\,\}$$

$$\{\,A[e/x]\,\}\ x\ \texttt{:=}\ e\ \{\,A\,\}$$

$$\frac{\{\,A\,\}\ c_1\ \{\,B\,\} \quad \{\,B\,\}\ c_2\ \{\,C\,\}}{\{\,A\,\}\ c_1\ \texttt{;}\ c_2\ \{\,C\,\}}$$

$$\frac{A' \Rightarrow A \quad \{\,A\,\}\ c\ \{\,B\,\} \quad B \Rightarrow B'}{\{\,A'\,\}\ c\ \{\,B'\,\}}$$

$$\frac{\{\,A \wedge b\,\}\ c_1\ \{\,B\,\} \quad \{\,A \wedge \neg b\,\}\ c_2\ \{\,B\,\}}{\{\,A\,\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{\,B\,\}}$$

$$\frac{\{\,A \wedge b\,\}\ c\ \{\,A\,\}}{\{\,A\,\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{\,A \wedge \neg b\,\}}$$

THE UNIVERSITY
of LIVERPOOL

Appendix C: OBJ Semantics

```
*** the programming language: expressions ***
obj EXP is pr ZZ .
           pr QID *(sort Id to Var) .
  sort  Exp.
  subsorts  Var Int < Exp .
  op  _+_  : Exp Exp -> Exp [prec 10] .
  op  _*_  : Exp Exp -> Exp [prec 8] .
  op   -_  : Exp -> Exp .
  op  _-_  : Exp Exp -> Exp [prec 10] .
endo

obj TST is pr EXP .
  sort Tst .
  subsort  Bool < Tst .
  op  _<_ : Exp Exp -> Tst [prec 15] .
  op  _<=_ : Exp Exp -> Tst [prec 15] .
  op  _is_ : Exp Exp -> Tst [prec 15] .
  op  not_ : Tst -> Tst [prec 1] .
  op  _and_ : Tst Tst -> Tst [prec 20] .
  op  _or_ : Tst Tst -> Tst [prec 25] .
endo

*** the programming language: basic programs ***
obj BPGM is pr TST .
  sort  BPgm .
  op  _:=_  : Var Exp -> BPgm [prec 20] .
endo
```

```
*** semantics of basic programs ***
th STORE is pr BPGM .
  sort Store .
  op initial : -> Store .
  op  _[[_]] : Store Exp -> Int [prec 65] .
  op  _[[_]] : Store Tst -> Bool [prec 65] .
  op     _;_ : Store BPgm -> Store [prec 60] .
  var  S : Store .
  vars X1 X2 : Var .
  var  I : Int .
  vars E1 E2 : Exp .
  vars T1 T2 : Tst .
  var  B : Bool .

  eq  initial [[X1]]  =  0 .

  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 ***
obj PGM is pr BPGM .
  sort  Pgm .
  subsort  BPgm < Pgm .
  op  skip  : -> Pgm .
  op  _;_   : Pgm Pgm -> Pgm [assoc prec 50] .
  op  if_then_else_fi : Tst Pgm Pgm -> Pgm [prec 40] .
  op  while_do_od     : Tst Pgm -> Pgm [prec 40] .
endo
```

THE UNIVERSITY
of LIVERPOOL

```
th SEM is pr PGM .
          pr STORE .
  sort EStore .
  subsort Store < EStore .
  op  _;_ : EStore Pgm -> EStore [prec 60] .
  var S : Store .
  var T : Tst .
  var P1 P2 : Pgm .
  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
```