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

THE UNIVERSITY
of LIVERPOOL

MAY 2006 EXAMINATIONS

Bachelor of Arts : Year 3
Bachelor of Engineering : Year 3
Bachelor of Science : Year 3
Doctor in Philosphy : Year 2
Master of Engineering : Year 2
Master of Philosphy : Year 2
No qualification aimed for : Year 1

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. 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. **[7 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 $[\![E]\!]_{\text{Exp}}$ for arithmetic 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:

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

   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 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 an assignment. **[10 marks]**

      ii. Modify the definition of $[\![V := E]\!]_{\text{Pgm}}$ to take account of the changes in the definition of $[\![E]\!]_{\text{Exp}}$. **[8 marks]**

2. Give detailed definitions for the following:

   (a) Equation and equational theory. **[9 marks]**

   (b) Term algebra **[7 marks]**

   (c) Term rewriting **[9 marks]**

3. The algebraic specification in Appendix B defines the semantics of a simple programming language. We can use the denotational semantics given in Appendix A to construct a model D of this specification, where $D_{\texttt{Exp}}$ is the set of well-formed expressions in the language, $D_{\texttt{Tst}}$ is the set of well-formed boolean expressions, $D_{\texttt{Pgm}}$ is the set of well-formed programs, $D_{\texttt{Store}}$ is the set of functions $Var \to Int$ (i.e., the set of functions that take a variable and return an integer), and $D_{\texttt{EStore}}$ is the set $D_{\texttt{Store}} \cup \{\bot\}$, (i.e., a new element, $\bot$ is added to represent the 'undefined' store). In this model, the operator

```
_;_ : Store Pgm -> EStore
```

is interpreted as the function $D_{\_;\_} : D_{\texttt{Store}} \times D_{\texttt{Pgm}} \to D_{\texttt{EStore}}$ where for all $S \in D_{\texttt{Store}}$ and $P \in D_{\texttt{Pgm}}$,

$$D_{\_;\_}(S, P) = [\![P]\!]_{\texttt{Pgm}}(S) \ .$$

(a) Carry on the definition of D by defining the functions that interpret the following operations:

i. `_:=_ : Var Exp -> Pgm .` **[5 marks]**

ii. `_[[_]] : Store Exp -> Int .` **[5 marks]**

iii. `_[[_]] : Store Tst -> Bool .` **[5 marks]**

(b) Regarding the equations of the specification in Appendix B, what do we need to show in order to conclude that D is a model of the specification? Illustrate your answer with reference to the following equation:

```
var S : Store .
var X : Var .
var E : Exp .
eq  S ; X := E [[X]]  =  S [[E]] .
```

[10 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 $2^{\texttt{'y}}$:

```
'x := 1 ;  'i := 0;
while not('i is 'y)
do
  'x := 2 * 'x ;
  'i = 'i + 1
od
```

(a) Give a suitable precondition and postcondition to specify that the program sets `'x` to the value of $2^{\texttt{'y}}$. (OBJ notation for exponentiation is `_**_`.) **[5 marks]**

(b) Give a suitable invariant for the loop, which will allow you to prove the correctness of the program. **[10 marks]**

(c) Give an OBJ proof score that will prove the correctness of the program. **[10 marks]**

5. Linked lists of integers are a data structure used to store sequences of integers. A linked list is either empty ('`null`') or contains both an integer value (the '`head`' of the list) and another linked list (the '`tail`' of the list). An abstract data type of linked lists can be specified in OBJ as follows:

```
obj LINKED_LIST is pr INT.

  sort LList .
  op null : -> LList .
  op head : LList -> Int .
  op tail : LList -> LList .
  op add : Int LList -> LList .

  var L : LList .
  var I : Int .

  eq  head(add(I,L))  =  I .
  eq  tail(add(I,L))  =  L .
endo
```

Suppose we wanted to extend the language described in Appendix B with a data type of linked lists, so that we could write programs like the following, that computes the sum of all the values in a linked list `list`:

```
'count := 0;
while not isNull(list)
do
  'count := 'count + head(list);
  list := tail(list)
od
```

Here, `list` is a linked-list variable. We can add this to the language of Appendix B by adding the following declarations:

```
sort LLVar .
op list : -> LLVar .
```

We would also need a sort for the linked-list expressions that could be assigned to this variable.

(a) Give further OBJ declarations of sorts (e.g., linked-list expressions) and operations (`head`, `tail`, `isNull` and `_:=_`) that will allow programs like the one above in the language. **[10 marks]**

(b) Give OBJ equations that describe the semantics of these new constructs, using a new operation `_[[_]]` that takes a `Store` and a linked-list expression, and returns a `LList`. **[15 marks]**

THE UNIVERSITY
of LIVERPOOL

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
- $[\![\texttt{not } T]\!]_{\text{Tst}}(S) = \neg\, [\![T]\!]_{\text{Tst}}(S)$
- $[\![T_1 \texttt{ and } T_2]\!]_{\text{Tst}}(S) = [\![T_1]\!]_{\text{Tst}}(S) \wedge [\![T_2]\!]_{\text{Tst}}(S)$
- $[\![T_1 \texttt{ or } T_2]\!]_{\text{Tst}}(S) = [\![T_1]\!]_{\text{Tst}}(S) \vee [\![T_2]\!]_{\text{Tst}}(S)$
- $[\![\texttt{skip}]\!]_{\text{Pgm}}(S) = S$
- $[\![X \texttt{ := } E]\!]_{\text{Pgm}}(S) = S[X \leftarrow [\![E]\!]_{\text{Exp}}(S)]$
- $[\![P_1 \texttt{ ; } P_2]\!]_{\text{Pgm}}(S) = [\![P_2]\!]_{\text{Pgm}}([\![P_1]\!]_{\text{Pgm}}(S))$
- 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 $[\![\texttt{while}\ T\ \texttt{do}\ P\ \texttt{od}]\!]_{\text{Pgm}} = [\![\texttt{while}\ T\ \texttt{do}\ P\ \texttt{od}]\!]_{\text{Pgm}}([\![P]\!]_{\text{Pgm}}(S))$

Appendix B: 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
```

THE UNIVERSITY
of LIVERPOOL

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

```
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
```