UNIVERSITY OF
LIVERPOOL

MAY EXAMINATIONS 2008

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. The syntax and denotational semantics of a simple programming language are summarised in Appendix A. Suppose we want to generalise the assignments in that language to 'parallel assignments'; i.e., programs of the form

$$x_1 \texttt{ := } e_1 \texttt{ \& } x_2 \texttt{ := } e_2 \texttt{ \& } \cdots \texttt{ \& } x_n \texttt{ := } e_n$$

where each assignment is done in parallel rather than sequentially. This means that all the expressions e_i are evaluated in the same state, and their values stored in the corresponding variables x_i. For example, the program

```
'x := 'y & 'y := 'x
```

has the effect of swapping the values stored in the variables `'x` and `'y`. In case a variable occurs more than once in the list $x_1, \ldots, x_n$, all but the final assignment will be ignored. For example,

```
'x := 1 & 'y := 2 & 'x := 2
```

has the same effect as

```
'y := 2 & 'x := 2
```

(a) Modify the BNF definition of programs from Appendix A to allow parallel assignments. (Parallel assignments should consist of one or more assignments to be executed in parallel; to allow for an arbitrary number of assignments, you will need to introduce a new syntactic class ⟨**AssignmentList**⟩.) **[8 marks]**

(b) Extend the denotational semantics given in Appendix A to give a semantics for parallel assignments. (Just as it was useful in part (a) to introduce a new syntactic class ⟨**AssignmentList**⟩, it will be useful here to introduce a new semantics function $[\![\_]\!] : \langle \texttt{AssignmentList} \rangle \times \mathit{Store} \to \mathit{Store}$.) **[10 marks]**

(c) Use your answer to part (b) to show that the program

```
'x := 0 & 'x := 1
```

sets `'x` to 1. **[7 marks]**

2. Give definitions for each of the following:

(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. 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 powers of 2. Specifically, it sets the variable `'x` to the value of $2^{'y}$:

```
'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 $2^{'y}$. (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 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]**

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 $[\![\texttt{while } T \texttt{ do } P \texttt{ od}]\!]_{\text{Pgm}}(S) = [\![\texttt{while } T \texttt{ do } P \texttt{ od}]\!]_{\text{Pgm}}([\![P]\!]_{\text{Pgm}}(S))$

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
```

```
*** 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
```