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

THE UNIVERSITY
*of* LIVERPOOL

# JANUARY 2004 EXAMINATIONS

Bachelor of Arts : Year 3
Bachelor of Engineering : Year 3
Bachelor of Science : Year 3
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. Appendix A summarises the syntax and denotational semantics of a simple imperative programming language. We want to extend this language with case conditionals of the form

```
case E of
  N1 : P1 ;
  N2 : P2 ;

  Nm : Pm
endcase
```

where $E$ is an arithmetic expression, each $N_i$ is a number, and each $P_i$ is a program. This program is executed by first evaluating the expression $E$ to obtain a number $N$; if the first occurrence of $N$ in the list $N_1, \ldots, N_m$ is $N_i$ (we allow that the list $N_1, \ldots, N_m$ may contain duplicates), then program $P_i$ is executed; if $N$ doesn't occur in the list $N_1, \ldots, N_m$ then the program immediately terminates (i.e., is equivalent to `skip`).

   (a) Give a BNF definition of the syntax of case conditionals. (Hint: it is helpful to use a separate syntactic class ⟨`CaseList`⟩ for the list of cases between '`of`' and '`endcase`'.) **[7 marks]**

   (b) Give a denotational semantics for case conditionals. (Hint: just as it was helpful to introduce a new syntactic class ⟨`CaseList`⟩ in defining the syntax of case conditionals, here it is useful to introduce a new semantic function

$$[\![cl]\!]_{\mathrm{CL}} : \mathit{Int} \times \mathit{Store} \to \mathit{Store} \,,$$

   such that for a CaseList $cl$, number $n$ and Store $S$, $[\![cl]\!]_{\mathrm{CL}}(n, S)$ gives the Store that results from choosing the first program in $cl$ with label $n$ and running it in state $S$. **[18 marks]**

2. Give definitions for each of the following:

   (a) Equational theory. **[6 marks]**

   (b) Term algebra. **[6 marks]**

   (c) Model of an equational theory. **[7 marks]**

   (d) Initial model of an equational theory. **[6 marks]**

3. In the lectures, we compared the denotational semantics of Appendix A to the algebraic semantics of Appendix B by showing that for any model, $A$, of the algebraic semantics, we could construct denotational functions

$$[\![P]\!]^A_{\text{Pgm}} : \mathit{Store} \to \mathit{Store} ,$$

etc. (recall that these were partial functions). This question asks you to relate the two semantics 'the other way round,' by showing that the denotational semantics allows us to construct a model of the algebraic semantics. This model, let's call it $D$, interprets the sort name `Store` as the set $\mathit{Var} \to \mathit{Int}$ of all functions from program variables to integers, and interprets the supersort `EStore` as the set consisting of all such functions from program variables to integers, plus a new element $\perp$. The idea is that $\perp$ represents the 'result' of a non-terminating computation; loosely, we say that

$$[\![P]\!]_{\text{Pgm}}(S) = \perp$$

if $[\![P]\!]_{\text{Pgm}}$ is undefined on the Store $S$.

Given that $D$ interprets `Store` and `EStore` as described above, sketch the remainder of the definition of $D$ as a model of the specification of Appendix B, by saying:

(a) how $D$ interprets the remaining sort names; **[5 marks]**

(b) how $D$ interprets the operations in `STORE`; **[10 marks]**

(c) showing that $D$ satisfies the 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 $2^{\text{'y}}$:

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

THE UNIVERSITY
*of* LIVERPOOL

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 .
  eq  head(null)  =  0 .
  eq  tail(null)  =  null .
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 `l`:

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

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

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

(a) Give further OBJ declarations of sorts 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
```

```
*** 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
```