THE UNIVERSITY
of LIVERPOOL

JANUARY 2005 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. The syntax and denotational semantics of a simple programming language are summarised in Appendix A. Suppose we want to extend the syntax of arithmetic expressions with a post-increment operator, to include expressions of the form $V{+}{+}$, where V is a variable (i.e., of syntactic class $\langle$**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 arithmetic expressions that includes expressions of the form $V{+}{+}$. **[4 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

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

   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 E is of the form $V{+}{+}$. **[10 marks]**

   ii. Modify the definition of $[\![V := E]\!]_{\text{Pgm}}$ 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. 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. (a) Say what is meant by 'term-rewriting'. **[10 marks]**

   (b) Say why term-rewriting is sound with respect to equational satisfaction. **[15 marks]**

4. The following OBJ specification defines the factorial function on integers.

```
obj FACTORIAL is
  pr ZZ .

  op  fac : Int -> Int .

  var I : Int .

  cq  fac(I)  =  1               if  I <= 0 .
  cq  fac(I)  =  I * fac(I - 1)  if  I > 0 .
endo
```

The following program sets the variable `'x` to the factorial of the value stored in `'y`:

```
'x := 1 ;
'count := 0 ;
while 'count < 'y
do
  'count := 'count + 1 ;
  'x := 'count * 'x
od
```

(a) Write an OBJ module that gives pre- and post-conditions that state that the program sets `'x` to the factorial of the value initially stored in `'y`. **[8 marks]**

(b) Give an invariant that will allow you to prove the partial correctness of the program. **[8 marks]**

(c) Give an OBJ proof score that proves the partial correctness of the program. **[9 marks]**

5. An abstract data type of pairs of integers is given in the following OBJ specification:

```
obj PAIR is
  pr ZZ .

  sort Pair .

  op <_,_> : Int Int -> Pair .
  ops (fst_) (snd_) : Pair -> Int .

  vars I J : Int .

  eq  fst < I , J >  =  I .
  eq  snd < I , J >  =  J .

endo
```

We want to extend the programming language described in Appendix B with a data type of pairs, so that we can write programs such as the following:

```
q := < 1 , 2 > ;  (p).1 := (q).2 ;  (p).2 := (q).1
```

where `p` and `q` are variables of the programming language, `(_).1` and `(_).2` refer to the first and second components of a pair, `<E1,E2>` represents a pair whose first component is the value of the integer expression `E1` and whose second component is the value of the integer expression `E2`, and the overloaded operator `_:=_` allows assignments either to a 'pair variable' such as `p` or `q`, or to a component of a pair variable. This program sets `q` to a pair whose first component is `1` and whose second component is `2`, then sets the first component of `p` to the second component of `q` (i.e., the value `2`), and finally sets the second component of `p` to the first component of `q`. After the program has run, `q` has the value `<1,2>` and `p` has the value `<2,1>`.

(a) Specify the syntax of the extended language by completing the following OBJ specification with subsort and operator declarations (one of the overloaded assignment operators has been declared for you).

```
obj PAIR-PROGRAMS is ex PGM .

  *** Variables of the programming language:
  sort PairVar .
  ops p q : -> PairVar .

  *** First and second components of pairs:
  sort PairComponent .

  *** Expressions of type Pair:
  sort PairExp .

  *** Subsort declarations:

  *** Operations of the language:
  op _:=_ : PairComponent Exp -> BPgm .

endo
```

[7 marks]

(b) The semantics of the extended language can be specified by overloading the operator `_[[_]]` as in the following OBJ module:

```
th PAIR-SEMANTICS is pr SEM .
                     pr PAIR .
                     pr PAIR-PROGRAMS .

  op _[[_]] : Store PairExp -> Pair .

endth
```

Define the semantics of the extended language by giving suitable equations to include in `PAIR-SEMANTICS`. **[12 marks]**

(c) Use the equations in your answer to part (b) to simplify the following term:

```
(s ; q := <1,2> ; (p).1 := (q).2 ; (p).2 := (q).1)[[p]]
```

for a given Store `s`. **[6 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
```

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
```