UNIVERSITY OF LIVERPOOL

MAY EXAMINATIONS 2009

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. 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 expression, each N_i is an integer, and each P_i is a program. This program is executed by first evaluating the expression E to obtain an integer 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`).

For example, the program

```
case  'x + 1  of
  0 : 'z := 5 ;;
  3 : 'z := 6 ;;
  4 : 'y := 0
endcase
```

will set `'z` to 5 if `'x` has the value -1; it will set `'z` to 6 if `'x` has the value 2; it will set `'y` to 0 if `'x` has the value 3; and it will have no effect if `'x` has any other value.

(a) i. Give a BNF definition of a syntactic category ⟨`CaseList`⟩ for the list of cases, where the list either consists of a single case, of the form N : P, with N an integer and P a program, or is of the form N : P `;;` CL, where CL is a CaseList. For example,

`0 : 'z := 5  ;;  3 : 'z := 6  ;;  4 : 'y := 0`

is a ⟨`CaseList`⟩. **[5 marks]**

ii. Extend the BNF syntax of the programming language with a clause stating that Programs (⟨Pgm⟩) may also consist of case conditionals of the form

`case` E `of` CL `endcase`

where E is an expression and CL a CaseList. **[2 marks]**

(b) i. Define a semantic function for CaseLists

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

such that for a CaseList CL, integer 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. For example, it should follow from your definition that

$$[\![\mathtt{0} :' \mathtt{z} := \mathtt{5}\ ;;\ \mathtt{3} :' \mathtt{z} := \mathtt{6}\ ;;\ \mathtt{4} :' \mathtt{y} := \mathtt{0}]\!]_{\mathrm{CL}}(3, S)$$

will return the state that results from running the program `'z := 6` in the Store S. **[8 marks]**

ii. Extend the definition of $[\![\ ]\!]_{\text{Pgm}}$ given in Appendix A to give a semantics for case conditionals; i.e., define

$$[\![\texttt{case}\ E\ \texttt{of}\ CL\ \texttt{endcase}]\!]_{\text{Pgm}}$$

where E is an expression and CL a CaseList. **[6 marks]**

iii. Use your answers to parts (i) and (ii) to calculate the semantics of the following program:

```
'x := 2 ;
case  'x + 1  of
  0 : 'z := 5 ;;
  3 : 'z := 6 ;;
  4 : 'y := 0
endcase
```

[4 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. Describe term rewriting in detail, illustrating the process with a simple example of an OBJ specification of natural numbers and arithmetic operations such as addition and multiplication. **[25 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) Briefly describe what it means for a program to be correct with respect to a given pre- and post-condition, and say why invariants can be used to prove the correctness of programs. **[6 marks]**

(b) Write an OBJ module that gives pre- and post-conditions that state that the above program sets 'x to the factorial of the value initially stored in 'y. **[6 marks]**

(c) Give an invariant that will allow you to prove the partial correctness of the program. **[6 marks]**

(d) Give an OBJ proof score that proves the correctness of the program. **[7 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, such as (p).1 or (p).2.. 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]**

Appendix A: The Language and its Semantics

Syntax

$\langle$Exp$\rangle$ ::= $\langle$Num$\rangle$ | $\langle$Var$\rangle$ | $\langle$Exp$\rangle$ + $\langle$Exp$\rangle$ | $\langle$Exp$\rangle$ - $\langle$Exp$\rangle$ | $\langle$Exp$\rangle$ * $\langle$Exp$\rangle$

$\langle$Tst$\rangle$ ::= true | false | $\langle$Exp$\rangle$ is $\langle$Exp$\rangle$ | $\langle$Exp$\rangle$ < $\langle$Exp$\rangle$
| $\langle$Tst$\rangle$ and $\langle$Tst$\rangle$ | $\langle$Tst$\rangle$ or $\langle$Tst$\rangle$ | not $\langle$Tst$\rangle$

$\langle$Pgm$\rangle$ ::= skip | $\langle$Var$\rangle$:= $\langle$Exp$\rangle$ | $\langle$Pgm$\rangle$; $\langle$Pgm$\rangle$
| if $\langle$Tst$\rangle$ then $\langle$Pgm$\rangle$ else $\langle$Pgm$\rangle$ fi
| while $\langle$Tst$\rangle$ do $\langle$Pgm$\rangle$ 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
- $[\![\texttt{not } T]\!]_{\text{Tst}}(S) = \neg\, [\![T]\!]_{\text{Tst}}(S)$
- $[\![T_1 \texttt{ and } T_2]\!]_{\text{Tst}}(S) = [\![T_1]\!]_{\text{Tst}}(S) \wedge [\![T_2]\!]_{\text{Tst}}(S)$
- $[\![T_1 \texttt{ or } T_2]\!]_{\text{Tst}}(S) = [\![T_1]\!]_{\text{Tst}}(S) \vee [\![T_2]\!]_{\text{Tst}}(S)$
- $[\![\texttt{skip}]\!]_{\text{Pgm}}(S) = S$
- $[\![X \texttt{ := } E]\!]_{\text{Pgm}}(S) = S[[\![E]\!]_{\text{Exp}}(S)/X]$
- $[\![P_1 \texttt{ ; } P_2]\!]_{\text{Pgm}}(S) = [\![P_2]\!]_{\text{Pgm}}([\![P_1]\!]_{\text{Pgm}}(S))$
- 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
```