Second Semester Examinations 2012/13

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. Sometimes, programs are only expected to work under certain conditions. For example, a program that uses division will only work when the divisor is non-zero. We might call a state in which the divisor is zero an 'error state'. Some programming languages, such as Java, allow the programmer to signal error states, and to execute 'error-recovery code' when such states arise (in Java, this is done by throwing and catching Exceptions). This question asks you to extend the language of Appendix A ('SIMPLE') with exceptions and error-recovery.

We might introduce a new command that throws an exception, and provides an error message as a String. Specifically, we extend the language with a new form of program:

`throwException(`M`)`

where M is a string. Exceptions are 'caught' by means of programs of the form:

`try` P_1 `catchException` P_2 `endCatch`

where P_1 and P_2 are programs: we call P_1 the 'try-block' and P_2 the 'catch-block'. The effect of `throwException` is to halt execution of the program at that point. If this is done within a try-block, then the catch-block is executed (in the state as it was before the exception was thrown); if this is not done within a try-block, the whole program immediately terminates. For example, here's a program that throws and catches an exception:

```
try
    'x := 1 ;
    if 'x == 1 then throwException("error") else skip fi ;
    'x := 5
catchException
    'x := 'x * 2
endCatch ;
'y := 'x
```

This program sets `'x` to 1, then throws an exception with error message `"error"`. The assignment `'x := 5` is *not* executed, but flow-of-control goes to the catch block, where the value of `'x` is doubled, so `'y` ends up with the value 2. However, if we run *this* program:

```
try
    'x := 2 ;
    if 'x == 1 then throwException("error") else skip fi ;
    'x := 5
catchException
    'x := 'x * 2
endCatch ;
'y := 'x
```

then `'x` starts off with the value 2, so the condition `'x == 1` is false, so no exception is thrown. Then `'x` is set to 5, the catch-block is *not* executed, and `'y` ends up with the value 5.

(a) Extend the BNF syntax of SIMPLE to include the constructs described above for throwing and catching exceptions (you may assume a syntactic category of strings has been provided). **[4 marks]**

(b) Extend the syntax of SIMPLE in Maude: say what operation declarations would need to be added to the module `PROGRAMS` in Appendix B. **[4 marks]**

(c) This addition to the language requires the semantics given in Appendix A to be completely rewritten. The denotational semantics of SIMPLE now requires two different sorts of states: states in the old sense (functions from variables to integers); and 'exception states'. Exception states are pairs (M, S), where M is a string (the error message), and S is a state (function from variables to integers). This state is the one that will be used in the catch-block, if there is one. The type of the denotation function $[\![\_]\!]_{\mathrm{Pgm}}$ is therefore

$$Program \times State^{+} \rightarrow State^{+}$$

where $State^{+} = State \cup String \times State$. An inductive definition for this function has to have equations that define $[\![P]\!]_{\mathrm{Pgm}}$ when the input is a state S, and when the input is an exception state (M, S). In other words, we need to define both $[\![P]\!]_{\mathrm{Pgm}}(S)$ and $[\![P]\!]_{\mathrm{Pgm}}(M, S)$ for all programs P, states S, and strings M.

i. Give an equation that says that when any program is run in an exception state, the result is the same exception state. (I.e., when an exception is thrown, any following program code is not executed.) **[2 marks]**

ii. Give an equation that says that the effect of `throwException(`M`)`, when run in a state S, is to create the exception state (M, S). **[2 marks]**

iii. For any programs P_1 and P_2, and state S, define
$[\![\texttt{try}\ P_1\ \texttt{catchException}\ P_2\ \texttt{endCatch}]\!]_{\mathrm{Pgm}}(S)$.
(*Hint*: $[\![P_1]\!]_{\mathrm{Pgm}}$ returns either a state or an exception state.) **[3 marks]**

iv. Complete the inductive definition of $[\![\_]\!]_{\mathrm{Pgm}}$. **[2 marks]**

v. Use your answers to the above to show that the following program sets `'a` to 10.

```
try
    'a := 0 ;
    try
        throwException("error1") ;
        'a := 5
    catchException
        'a := 'a + 1 ;
    endCatch ;
    'a := 'a + 9
catchException
    'a := 'a + 1
endCatch
```

[5 marks]

vi. the module `STORE` in Appendix B declares a sort `Store` that represents states. How might you use subsorts and operations in Maude to specify a sort `State+` that represents $State^{+}$ (i.e., a sort that contains both states *and* exception states)? **[3 marks]**

2. Consider the following Maude specification.

```
fmod  ARITHMETIC  is

  sort  Number .

  op  0 : -> Number .
  op  succ : Number -> Number .
  op  plus : Number Number -> Number .

  vars  M N : Number .

  eq  plus(0, N)  =  N .
  eq  plus(succ(M), N)  =  succ(plus(M, N)) .

endfm
```

(a) Give a general definition of *signature*. **[2 marks]**

(b) Say what the signature of `ARITHMETIC` is. **[3 marks]**

(c) For a signature $\Sigma = (S, O)$, give a general definition of Σ-*terms*. **[3 marks]**

(d) Give an explicit definition of Σ-terms where Σ is the signature of `ARITHMETIC`. **[2 marks]**

(e) For a signature $\Sigma = (S, O)$, give a general definition of a Σ-*model*. **[3 marks]**

(f) For a signature $\Sigma = (S, O)$, give a general definition of the Σ-*term algebra*, T_Σ. **[3 marks]**

(g) Give two examples of Σ-models (other than the term algebra) where Σ is the signature of `ARITHMETIC`. **[4 marks]**

(h) For a signature Σ and any Σ-model A, there is a homomorphism $h : T_\Sigma \to A$. Give the definition of this homomorphism. **[3 marks]**

(i) For each of the two models in your answer to part (g), give the result of applying the homomorphism from part (h) to the term `plus(succ(0),succ(0))`. **[2 marks]**

3. **(a)** For an arbitrary signature Σ, give a general definition of Σ-*equation*. **[3 marks]**

(b) Say what it means for a Σ-model to *satisfy* a Σ-equation. **[3 marks]**

(c) Describe the process of term-rewriting, and say how a set E of Σ-equations defines a relation $\overset{*}{\leftrightarrow}_E$ on Σ-terms, where $t \overset{*}{\leftrightarrow}_E t'$ if and only if t can be rewritten to t' using the equations in E. **[5 marks]**

(d) Illustrate your answer to part (c) be describing how the term `plus(succ(succ(0)),succ(0))` can be rewritten using the equations in the module `ARITHMETIC` from Question 2 **[4 marks]**

(e) Describe how the relation $\overset{*}{\leftrightarrow}_E$ allows the construction of a (Σ, E)-model, the quotient term algebra, T_Σ/E. **[5 marks]**

(f) What is meant by an *initial* (Σ, E)-*model*? **[2 marks]**

(g) Briefly say why T_Σ/E is an initial (Σ, E)-model. **[3 marks]**

4. The following program, written in the language specified in Appendix B, sets the variable `'z` to the square of the value of `'n` (provided that this value is at least 0), without using multiplication.

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

(a) Simplify the following terms (assuming `s` is a `Store`), where *body* is the program

```
'x := 'x + 1 ; 'z := 'z + 'y + 1 ; 'y := 'y + 2 .
```

i. `s ;` *body* `[[ 'x ]]` **[2 marks]**

ii. `s ;` *body* `[[ 'y ]]` **[2 marks]**

iii. `s ;` *body* `[[ 'z ]]` **[2 marks]**

(b) In general, what is meant by an 'invariant' of a loop? **[3 marks]**

(c) Consider the predicate `invy`:

```
op  invy  : Store -> Bool .
var S : Store .
eq  invy(S)  =  (S[[ 'y ]]) == 2 * (S[[ 'x ]]) .
```

Say why `invy` is an invariant of the loop in the program above. **[3 marks]**

(d) Give a pre- and post-condition that specify that `'z` should be set to the square of the value of `'n` (provided that this value is at least 0) **[4 marks]**

(e) Give an invariant that would allow you to prove that the program above is correct with respect to your answer to part (d). **[4 marks]**

(f) Sketch how you would use Maude to prove that the program is correct. **[5 marks]**

5. A *2-register machine* is an abstract machine that has two 'registers', `r1` and `r2`, that each store an integer value. The values in these registers are updated by programs, which are sequences of instructions. The basic instructions are as follows.

- `inc1` and `inc2`, which add 1 to the value stored in `r1` and `r2`, respectively; i.e., `inc1` adds 1 to the value stored in `r1` and leaves the value in `r2` unchanged, and similarly for `inc2`.
- `dec1` and `dec2`, which subtract 1 from the value stored in `r1` and `r2`, respectively.
- `copy`, which sets the value of `r2` to the value of `r1`, and sets the value of `r1` to 0.

Programs are sequences of instructions, with a loop construct of the form

`while r2>0` $\{\ P\ \}$

where P is a program. This repeatedly executes P while the value stored in `r2` is greater than 0; if the value stored in `r2` is less than or equal to 0, the program does nothing. For example, the following program, which we'll call `double`, doubles the value stored in `r1`.

```
copy
while r2>0 {
   dec2
   inc1
   inc1
}
```

(a) Give a BNF specification of programs. **[3 marks]**

(b) Give a Maude specification of programs, using a sort `Program`, so that 2-register programs are terms of sort `Program`. **[4 marks]**

(c) Briefly say why `double` is a well-formed term of sort `Program`. **[2 marks]**

(d) Give a Maude specification of the semantics of programs. (*Hint*: the state of a 2-register machine is just a pair of integers; specify pairs of integers and give equations describing the effects of programs on these pairs.) **[10 marks]**

(e) Use your answer to part (d) to show that, when `double` is run in a state where the value in `r1` is 2, it results in a state where `r1` stores the value 4. **[6 marks]**

Appendix A: The Language SIMPLE 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`$\rangle$

$\langle$`BExp`$\rangle$ `::= true | false |` $\langle$`Exp`$\rangle$ `==` $\langle$`Exp`$\rangle$ `|` $\langle$`Exp`$\rangle$ `<` $\langle$`Exp`$\rangle$
`|` $\langle$`BExp`$\rangle$ `and` $\langle$`BExp`$\rangle$ `|` $\langle$`BExp`$\rangle$ `or` $\langle$`BExp`$\rangle$ `| not` $\langle$`BExp`$\rangle$

$\langle$`Pgm`$\rangle$ `::= skip |` $\langle$`Var`$\rangle$ `:=` $\langle$`Exp`$\rangle$ `|` $\langle$`Pgm`$\rangle$ `;` $\langle$`Pgm`$\rangle$
`| if` $\langle$`BExp`$\rangle$ `then` $\langle$`Pgm`$\rangle$ `else` $\langle$`Pgm`$\rangle$ `fi`
`| while` $\langle$`BExp`$\rangle$ `do` $\langle$`Pgm`$\rangle$ `od`

Summary of the Denotational Semantics

- $[\![N]\!]_{\text{Exp}}(S) = N$
- $[\![V]\!]_{\text{Exp}}(S) = S(V)$
- $[\![E_1 \texttt{ + } E_2]\!]_{\text{Exp}}(S) = [\![E_1]\!]_{\text{Exp}}(S) + [\![E_2]\!]_{\text{Exp}}(S)$
- $[\![E_1 \texttt{ - } E_2]\!]_{\text{Exp}}(S) = [\![E_1]\!]_{\text{Exp}}(S) - [\![E_2]\!]_{\text{Exp}}(S)$
- $[\![E_1 \texttt{ * } E_2]\!]_{\text{Exp}}(S) = [\![E_1]\!]_{\text{Exp}}(S) * [\![E_2]\!]_{\text{Exp}}(S)$
- $[\![\texttt{true}]\!]_{\text{Tst}}(S) = \mathit{true}$
- $[\![\texttt{false}]\!]_{\text{Tst}}(S) = \mathit{false}$
- $[\![E_1 \texttt{ == } E_2]\!]_{\text{Tst}}(S) = v$, 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: Maude Semantics of SIMPLE

```
*** the programming language: expressions ***
fmod EXPRESSION is pr INT .
                   pr QID *(sort Id to Variable) .
  sort  Expression.
  subsorts  Variable Int < Expression .
  op  _+_  : Expression Expression -> Expression .
  op  _*_  : Expression Expression -> Expression .
  op   -_  : Expression -> Expression .
  op  _-_  : Expression Expression -> Expression .
endfm

fmod BOOLEAN-EXPRESSION is pr EXPRESSION .
  sort BooleanExpression .
  ops  true false : -> BooleanExpression .
  op  _<_ : Expression Expression -> BooleanExpression .
  op  _<=_ : Expression Expression -> BooleanExpression .
  op  _is_ : Expression Expression -> BooleanExpression .
  op  not_ : BooleanExpression -> BooleanExpression .
  op  _and_ : BooleanExpression BooleanExpression -> BooleanExpression .
  op  _or_ : BooleanExpression BooleanExpression -> BooleanExpression .
endfm

*** the programming language: basic programs ***
fmod BASIC-PGM is pr BOOLEAN-EXPRESSION .
  sort  BasicProgram .
  op  _:=_  : Variable Expression -> BasicProgram .
endfm
```

```
*** semantics of basic programs ***
th STORE is pr BASIC-PGM .
  sort Store .

  op  _[[_]] : Store Expression -> Int .
  op  _[[_]] : Store BooleanExpression -> Bool .
  op     _;_ : Store BasicProgram -> Store .
  var  S : Store .
  vars X1 X2 : Variable .
  var  I : Int .
  vars E1 E2 : Expression .
  vars T1 T2 : BooleanExpression .
  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 == E2]]  =  (S [[E1]]) == (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 ***
fmod PROGRAMS is pr BASIC-PGM .
  sort  Program .
  subsort  BasicProgram < Program .
  op  skip  : -> Program .
  op  _;_   : Program Program -> Program .
  op  if_then_else_fi : BooleanExpression Program Program -> Program .
  op  while_do_od     : BooleanExpression Program -> Program .
endfm
```

```
th SEM is pr PROGRAMS .
          pr STORE .
  sort ErrorStore .
  subsort Store < ErrorStore .
  op  _;_ : ErrorStore Program -> ErrorStore .
  var S : Store .
  var T : BooleanExpression .
  var P1 P2 : Program .
  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
```