FIRST SEMESTER EXAMINATIONS 2013/14

Advanced Object-Oriented Programming

TIME ALLOWED : Two 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. Below is a Maude specification of ‘search’ trees of integers. A search tree is a binary tree that is either empty (written as the constant ‘null’), or is of the form node(T1, I, T2), where I is an integer (we call it the ‘node value’ of the tree) and T1 and T2 are search trees (we call them the left and right subtrees of the tree), with the property that all the integers in the left subtree T1 are less than I, and all the integers in the right subtree T2 are greater than I — moreover, all the nodes in T1 and T2 have the same property: all integers in the left subtree are less than the node value, and all integers in the right subtree are greater than the node value.

```
mod SEARCH-TREES is
    protecting INT .
    sort SearchTree .
    op null : -> SearchTree [ ctor ] .
    op node : SearchTree Int SearchTree -> SearchTree [ ctor ] .
    op height : SearchTree -> Int .
    op insert : SearchTree Int -> SearchTree .
    vars I J : Int .
    vars T1 T2 : SearchTree .
    eq height(null) = 0 .
    eq height(node(T1, I, T2)) = max(height(T1), height(T2)) + 1 .
    eq insert(I, null) = node(null, I, null) .
    eq insert(I, node(T1, I, T2)) = node(T1, I, T2) .
    cq insert(I, node(T1, J, T2)) = node(insert(I, T1), J, T2) if I < J .
    cq insert(I, node(T1, J, T2)) = node(T1, J, insert(I, T2)) if I > J .
endm
```

Operations null and node are structural operations that provide a binary tree structure; it is possible to use these to construct trees that do not have the intended property that all integers in left subtrees are less than the node value and all integers in right subtrees are greater than the node value. For example,

```
node(node(null, 3, null), 2, node(null, 1, null))
```

does not have that property, as 3 is in the left subtree, but is not less than 2. However, the operation insert will ensure that this property holds: if it is called from a search tree, then the new value will be inserted at the appropriate point.

(a) Give a Java implementation of classSearchTree that uses a class, TreeNode, to provide the binary tree structure (in a similar way to how a class Node can be used to provide a linked list structure). Class TreeNode should also provide methods height() and insert(). Marks will be awarded for:

i. correct implementation of the binary tree structure using class TreeNode [3 marks]
ii. correct implementation of the constant `null` as a `SearchTree` constructor

iii. correct implementation of `node` as a `TreeNode` constructor

iv. correct implementation of `height()` in class `SearchTree`

v. correct implementation of `insert` in class `SearchTree`

vi. correct implementation of `height()` in class `TreeNode`

vii. correct implementation of `insert()` in class `TreeNode`

viii. appropriate use of scope modifiers.

(b) If class `TreeNode` were declared as a `protected` inner class within class `SearchTree`, where precisely would `TreeNode` and its public members be visible?
The Abstract Data Type of Count Lists is a list of pairs; each pair in the list has an integer (which we call the 'value') and another integer (which we call the 'count'); the idea is that the 'count' integer records how often the value occurs in the list. An example application might be to read in a sequence of integer values, perhaps recording the grades of individuals in a class, e.g.,

\[2, 7, 7, 5, 9, 3, 6, 7, 2, 5\]  \hfill (1)

and, for each grade value, record how often that grade occurs in the given data. For the example data above, we should have the following Count List (where pairs are grouped in brackets):

\[(2, 2), (3, 1), (5, 2), (6, 1), (7, 3), (9, 1)\].

Note that grades that don’t occur in the data, such as 4, are not listed.

As an Abstract Data Type, Count Lists have one constant, representing the empty list (no data); and the following operations:

- **add** a Pair (consisting of two integer values, a value and its count) to the list;
- **insert** a given integer value into the list (if it already occurs in the list, increment the count value; otherwise, add a new pair with the given value and a count of 1); and
- **getCount** — this takes an integer value as argument and returns the count of how often that value occurs in the list (0 if it doesn’t occur at all in the list).

Count Lists make use of a subsidiary Abstract Data Type of Pairs: this Abstract Data Type has three operations:

- a constructor that takes two integers and returns the Pair built from those values;
- **getValue**, returning the value component of the Pair; and
- **getCount**, returning the count.

**(a)** Specify the Abstract Data Type of Pairs in Maude.  \[6 \text{ marks}\]

**(b)** Specify Count Lists in Maude.  \[11 \text{ marks}\]

**(c)** Write a term in Maude that corresponds to the Count List recording all the data in Example (1) above.  \[3 \text{ marks}\]

**(d)** Sketch why the equations in your specification of Count Lists would allow you to conclude that the value 5 occurs twice in that list.  \[5 \text{ marks}\]
3. Consider the following class of binary trees. The method `isInOrder()` is intended to test whether the top label of the tree is greater than the top label of the left subtree and smaller than the top label of the left subtree.

```java
class BinaryTree
{
    private BinaryTree leftSubtree;
    private int label;
    private BinaryTree rightSubtree;

    BinaryTree(BinaryTree left, int val, BinaryTree right)
    {
        leftSubtree = left;
        label = val;
        rightSubtree = right;
    }

    public int getLabel()
    {
        return label;
    }

    public boolean isInOrder()
    {
        return leftSubtree.getLabel() < label
                && label < rightSubtree.getLabel();
    }

    public static void main(String[] args)
    {
        BinaryTree bt = new BinaryTree(null, 5, null);
        bt = new BinaryTree(bt, 7, null);
        System.out.println(bt.isInOrder());
    }
}
```

(a) Briefly describe the function of the ‘method-call stack’ in the Java interpreter. 

[3 marks]

(b) What happens when the `main` method in the `BinaryTree` class is executed? Describe the state of the method-call stack during the execution of this `main` method. 

[5 marks]

(c) Briefly describe the differences between ‘checked’ and ‘unchecked’ exceptions. 

[4 marks]

(d) Write a checked exception class, `NoSubtreeException`, and modify the `isInOrder()` method so that it throws a `NoSubtreeException` if either the left or right subtree is null. What other changes would be necessary to the class? 

[6 marks]
(e) What is meant by a ‘class invariant’? 

(f) Modify the constructor of class BinaryTree so that it throws a checked exception if either:

- the parameter \texttt{left} is not null and its label is not smaller than parameter \texttt{val}, or
- the parameter \texttt{right} is not null and its label is not greater than parameter \texttt{val}.

With this modification, is the property that \texttt{isInOrder()} always returns true a class invariant for BinaryTree?
4. Consider the following program, which is intended to print out the strings "a1", "a2", ...",a100", as well as the strings "b1", "b2", ..., "b100", using two threads, one to print all the ‘a’ strings, and one to print out all the ‘b’ strings. The printing of the ‘a’ and ‘b’ strings can therefore be interleaved.

```java
class ThreadTest implements Runnable {
    static String printString = null;
    private String threadName;

    ThreadTest(String s) { threadName = s; }

    public void run() {
        int i = 0;
        while (i < 100) {
            if (printString == null) {
                printString = threadName + (++i);
            } else {
                System.out.println(printString);
                printString = null;
            }
        }
    }
}

public static void main(String[] args) {
    Thread t1 = new Thread(new ThreadTest("a"));
    Thread t2 = new Thread(new ThreadTest("b"));
    t1.start();
    t2.start();
}
```

Each thread uses a local variable i as a count variable, and the static variable printString to store the strings to be printed. If printString is null, a thread will assign to it the next string to be printed, incrementing its count variable; if printString is not null, it will print out the string stored, and set printString to null, so it can be assigned the next string to be printed.

(a) What is meant by ‘interference’ in multi-threaded programs? [3 marks]
(b) Describe how interference might arise in this example. [6 marks]
(c) Describe how synchronization is used in Java to prevent interference. [8 marks]
(d) Briefly describe the difference between a synchronized method and a synchronized block of code. [3 marks]
(e) Describe how you could use a synchronized block of code in class ThreadTest in order to prevent interference. [5 marks]
5. Queues are First-In, First-Out lists: elements are removed from the ‘front’ of the queue and are added to the ‘end’ of the queue. A queue can be implemented by using an array, values, to store the elements in the queue, with an integer ‘pointer’, startIndex, indicating the index of the first element in the queue, and another, endIndex, whose value is one greater than the index of the last element in the queue. That is, the elements in the queue are all the values values[i], where startIndex ≤ i < endIndex.

Getting the element at the front of the queue can be done by returning the element stored in values[startIndex] and incrementing startIndex (or throwing an exception if the queue is empty, i.e., startIndex == endIndex).

Adding a value to the end of the queue is done by storing the value in values[endIndex] and incrementing endIndex, if endIndex is less than values.length. If endIndex is equal to values.length, then, if startIndex is greater than 0, room can be made to store the new element by moving all the values in the queue to the start of the array (and updating startIndex and endIndex). Otherwise, room will have to be made by creating a larger array (say, 10 more than values.length), copying all the elements in the queue to the larger array and assigning that array to the variable values, before storing the new element at the end.

(a) Implement a generic class Queue⟨E⟩, of homogeneous queues that store elements of the parameter type E. Marks will be awarded for:
   i. correct use of the parameter type E [2 marks]
   ii. implementation of the method getFirst() that returns the first element in the queue and removes it from the queue, as described above [5 marks]
   iii. implementation of a checked Exception class that is thrown by getFirst() if the queue is empty [3 marks]
   iv. implementation of the method addToEnd(E v) that adds the given (parameter) value to the end of the queue, as described above [9 marks]

(b) Generic types are implemented in Java by ‘erasure’. Briefly describe what erasure is, and illustrate your answer by giving the results of erasure on your addToEnd(E v) method from your answer to part (a) above [6 marks]