CS211 Fall 2003
Prelim 2 Solutions and Grading Guide

Problem 1:

(a) obj2 = obj1;

ILLEGAL because type of reference must always be a supertype of type of object

(b) obj3 = obj1;

ILLEGAL because type of reference must always be a supertype of type of object

(c) obj3 = obj2;

ILLEGAL because type of reference must always be a supertype of type of object

(d) I1 b = obj3;

LEGAL because C3 is a subclass of C1 which implements I1

(e) I2 c = obj1;

ILLEGAL because type of reference must always be a supertype of type of object

Grading (total 5 points):

For each part

-1 : wrong conclusion or reason

Problem 2(a):

```
abstract class Exp {
      abstract int eval();
}

class BinExp extends Exp {
      protected char op;
      protected Exp left;
      protected Exp right;

      public BinExp(char op, Exp l, Exp r) {
            this.op = op;
            this.left = l;
            this.right = r;
      }

      public int eval() {
            switch(op) {
                  case '+': return left.eval() + right.eval();
                  case '*': return left.eval() * right.eval();
                  default: System.out.println("ERROR: Unknown op");
                        return -1;
            }
      }

      public char get() { return op; }

      public Exp getLeft() { return left; }

      public Exp getRight() { return right; }
}

class NumExp extends Exp {
      protected int n;

      public NumExp(int n) { this.n = n; }

      public int get() { return n; }

      public int eval() { return n; }
}
```

Grading (total 10 points):

The solution for this part would vary widely. But at a minimum, a correct solution must have all the class definitions with variable declarations, constructors and getter methods. Setter methods are not required.

-7 : no separate class for numbers and binary operators
-4 : incorrect derivation of classes (e.g. NumExp should not be a subclass of BinExp)
-3 : BinExp class stores integers
-3 : NumExp class stores operators
-3 : no constructor for BinExp for directly setting left, right children
-3 : not enough getter methods

Problem 2(b):

```
public static int eval(Exp root) {
      if (root==null) {
            System.out.println("ERROR: Tree not initialized");
            return -1;
      }
      return root.eval();
}
```

Grading (total 10 points):

This part would greatly depend on the solution for part (a). At a minimum, it should implement a recursive method that evaluates the tree passed.

-2 : no error checking for `root == null`
-3 : does not work if root is just a NumExp node
-5 : illegal downcast if `eval()` implemented externally and Exp objects not checked for type before downcasting
-3 : returns wrong result
-2 : has any sort of parsing code (this problem does not require parsing expressions)

Problem 3(a):

$n, \quad n\log n, \quad n^2, \quad 2^n, \quad n!$ (in increasing order of asymptotic complexity)

Grading (total 7 points):

-2 : n not smallest
-2 : $n!$ not largest
-2 : n^2 smaller than n
-2 : $n\log n$ smaller than n

-2 : 2^n smaller than $n, \quad n\log n,$ or n^2
-2 : wrote in reverse order

Problem 3(b):

TRUE: $2^n = O(3^n)$ one valid witness pair: $(1,0)$

FALSE: $3^n = O(2^n)$
Proof: Assume $3^n = O(2^n)$. Therefore there exists a witness pair (c, n_0) such that $3^n \le c.2^n$ for all $n \ge n_0$. In other words:

$$\frac{3^n}{2^n} \le c \quad | \quad n \ge n_0$$

But the limit (as $n \to +\infty$) is $\frac{3^n}{2^n} = +\infty$. Therefore, it is not possible to have a constant upper bound on $\frac{3^n}{2^n}$. This implies our initial assumption of the existence of a witness pair was false. Therefore, the statement $3^n = O(2^n)$ is also false.

Grading (total 8 points):

-4 : first statement concluded FALSE
-2 : first statement concluded TRUE but invalid witness pair
-4 : second statement concluded TRUE
-2 : second statement concluded FALSE but no relevant argument (informal good enough)

Problem 3(c):

No. Here is a counter example:

Let $f(n) = 2n$ and $g(n) = n$. We can easily show that $f(n) = O(g(n))$ using the witness pair $(2,0)$. Now,

$$2^{f(n)} = 2^{2n} = 4^n \text{ and, } 2^{g(n)} = 2^n$$

By the same process that we used to show that $3^n = O(2^n)$ is false, we can prove that $4^n = O(2^n)$ is also false. Therefore, if $f(n) = O(g(n))$ it does not imply that $2^{f(n)} = O(2^{g(n)})$.

Grading (total 5 points):

-5 : wrong conclusion (answered yes instead of no)
-3 : if counter example (or other proof) not valid

Problem 4:

[Breadth-first]

a) ABDCE
b) Not unique. Another possibility: ADBCE

[Depth-first]

c) ABCED
d) Not unique. Another possibility: ADECB

e) Yes. Graph with one node (A) or, (A)→(B), or a graph that looks like a "linked list" in general, among many other possibilities.

Grading (total 10 points):

2 points for each part:

a) -2 if wrong sequence

b) -2 if answered "unique"
   -1 if answered "not unique" but gave wrong sequence

c) -2 if wrong sequence

d) -2 if answered "unique"
   -1 if answered "not unique" but gave wrong sequence

e) -2 if answered "no"
   -1 if answered "yes" but gave wrong example

Problem 5:

```
public static boolean Valid(String s) {
      if (s==null)
            return false;

      return Valid(s,0,s.length()-1);
}

public static boolean Valid(String s, int low, int high) {
      if (low > high)
            return true;
      if (low == high)
            return false;
      else
            return(s.charAt(low) == '(') &&
                  (s.charAt(high) == ')') &&
                  (Valid(s,low+1,high-1));
}
```

Grading (total 15 points):

-2 : function does not return Boolean
-2 : fails if s is null
-5 : does not work for empty string “”
-3 : *extremely* inefficient (e.g. scans string from beginning in each iteration)
-7 : does not work for strings of odd length (i.e. either crashes or returns true)
-2 : incorrect use of `s.charAt(i)`
-10 : no recursion
-3 : bad algorithm
-7 : allows invalid string
-1 : returns true if input is null

Problem 6:

```
class Hashley implements SearchStructure {
      protected ListCell[] spine;
      protected int size;
      private final int buckets = 10;

      public Hashley() {
            spine = new ListCell[buckets];
            for (int i=buckets; i<buckets; i++)
                  spine[i] = null;
      }

      public void insert(Object o) {
            int index = ((Integer) o).intValue() % buckets;

            ListCell l = new ListCell(o,spine[index]);
            spine[index] = l;
            ++size;
            return;
      }

      public void delete(Object o) {
            int index = ((Integer) o).intValue() % buckets;
            ListCell curr = spine[index];
            ListCell prev = null;

            while (curr != null &&
                  ((Comparable) curr.getDatum()).compareTo(o) != 0) {
                  prev = curr;
                  curr = curr.getNext();
            }

            if (curr == null)
                  return;

            if (prev == null)
                  spine[index] = curr.getNext();
            else
                  prev.setNext(curr.getNext());

            --size;
            return;
      }

      public boolean search(Object o) {
            int index = ((Integer) o).intValue() % buckets;
            ListCell curr = spine[index];

            while (curr != null) {
                  if (((Comparable) curr.getDatum()).compareTo(o) == 0)
                        return true;
                  curr = curr.getNext();
            }
            return false;
      }

      public int size() { return size; }
}
```

Grading (total 30 points):

-3 : class header does not have “implements SearchStructure”
-5 : spine is not declared as an array
-5 : spine array not allocated (no `new`) before first use
-3 : object not type-casted to `Integer` before calling `intValue()`
-2 : `insert()` does not increment size
-5 : deletion of first node in a list fails
-5 : deletion of intermediate nodes fail
-2 : `delete()` does not decrement size
-3 : objects not compared correctly
-5 : inefficient search if all lists are traversed to look for an object
-3 : tries to call methods on a null pointer (no checking in while loops etc.)
-3 : does not keep a `size` variable
-2 : each index in spine initialized to point to empty ListCell’s
-5 : function headers don’t match interface