CS 61B (Clancy, Yelick) Solutions and grading standards for exam 1 Spring 2001

Exam information

369 students took the exam. Scores ranged from 1 to 20, with a median of 11 and an average of 11.1. There were 40 scores between 15.5 and 20, 180 between 10.5 and 15, 132 between 5.5 and 10, and 17 between 2 and 5. (Were you to receive a grade of 16 on all your midterm exams, 48 on the final exam, plus good grades on homework and lab, you would receive an A–; similarly, a test grade of 11 may be projected to a B–.)

There were four versions of the exam, A, B, C, and D. (The version indicator appears at the bottom of the first page.) Versions A and C were identical except for the (reversed) order of the problems. Versions B and D were also identical except for the order of the problems.

If you think we made a mistake in grading your exam, describe the mistake in writing and hand the description with the exam to your lab t.a. or to Mike Clancy. We will regrade the entire exam.

Solutions and grading standards for versions A and B

Problem 0 (1 point)

You lost 1 point on this problem if you did any of the following:

- you earned some credit on a problem and did not put your name on the page,
- you did not indicate your lab section or t.a., or
- you failed to put the names of your neighbors on the exam.

The reason for this apparent harshness is that exams can get misplaced or come unstapled, and we would like to make sure that every page is identifiable. We also need to know where you will expect to get your exam returned. Finally, we occasionally need to know where students were sitting in the class room while the exam was being administered.

Problem 1 (2 points)

This was problem 4 on versions C and D. Version A asked for a statement that would initialize the joeAcct variable as shown in the diagram. Here are solutions:

```
Account joeAcct = new Account (100, new Account (500));
Account joeAcct = new Account (100, new Account (500, null));
```

Version B asked for essentially the same thing, except initializing maryAcct with different balances:

```
Account maryAcct = new Account (300, new Account (700));
Account maryAcct = new Account (300, new Account (700, null));
```

½ point was deducted for not declaring the variable, that is, omitting the first Account on the line, or for forgetting either new. 1 point was deducted for a correct two-statement solution; a two-statement solution with some other small error received ½ point out of 2. 1 point was also deducted or a solution with an incorrect attempt to initialize a new overdraft account, for example

```
Account maryAcct =
   new Account (300, Account parentAcct=new Account(700));
```

No credit was given for any solutions that did not explicitly initialize the overdraft account. For example,

```
Account maryAcct = new Account (300, myParent);
```

earned 0 out of 2.

Problem 2 (8 points)

This was problem 3 on versions C and D.

Part a, worth 2 points, involved identifying errors in a main program that tested the IslamicDate class. Answers are shown below.

```
public static void main (String [ ] args) {
   IslamicDate d1 = IslamicDate (1, 1);          error (new omitted; corrected by inserting new)
   IslamicDate d2 = d1.makeTomorrow ( );         error (a call to a void method is used where an IslamicDate expression is required)
   IslamicDate d3 = null;                        OK
   System.out.println ("" + d1);                 OK
   System.out.println (d3.tomorrow ( ));         error (d3 is null)
}
```

In version B, the fourth statement was moved to precede the assignment to d2; thus the answers were error, OK, error, OK, error. ½ point was deducted for each error, up to a maximum of 2.

In part b, worth 6 points, you were to write a method precedes (version A) or isLaterThan (version B) that could be used outside the IslamicDate class to compare two dates in the same year. (We explain only the precedes method here. isLaterThan is the same, except with appropriate comparisons reversed.) There are a variety of solution approaches, most resembling the daySpan computations of homework 2; examples appear below.

An iterative solution:

```
public static boolean precedes (IslamicDate d1, IslamicDate d2) {
   if (d1.equals (d2)) {
      return false;
   }
   IslamicDate d3;
   for (d3=d1.tomorrow ( );
        !d3.equals (d2) && !d3.equals (new IslamicDate (12,29));
        d3.makeTomorrow ( )) {
   }
   return d3.equals (d2);
}
```

A while loop or a rearranged for loop would also have been appropriate.

A recursive solution:

```
public static boolean precedes (IslamicDate d1, IslamicDate d2) {
   if (d1.equals (d2))
      return false;
   } else {
      return helper (d1, d2)
   }
}
public static boolean helper (IslamicDate d1, IslamicDate d2) {
   if (d1.equals (d2)) {
      return true;
   } else if (d1.equals (new IslamicDate (12,29))) {
      return false;
   } else {
      return helper (d1.tomorrow ( ), d2);
   }
}
```

A helper function or special case was needed to distinguish between two initially equal dates (for which false is returned) from dates found to be equal in the recursion base case (for which true is returned).

An alternative approach that could be implemented in either an iterative or a recursive method is to call .tomorrow ( ) on both dates, then to "race" them forward using .makeTomorrow ( ). This approach determines which date is later by seeing which one first reaches the date 1/1. A variation on this approach is to start two dates at 1/1, then count the number of calls to .makeTomorrow ( ) necessary to reach the argument dates.

A significantly different approach uses toString to access the month and day of each date:

```
public static boolean precedes (IslamicDate d1, IslamicDate d2) {
   String s1 = d1.toString ( );
   String s2 = d2.toString ( );
   int month1 = Integer.parseInt (s1.substring (0,s1.indexOf("/")));
   int day1 = Integer.parseInt (s1.substring (s1.indexOf("/")+1));
   int month2 = Integer.parseInt (s2.substring (0,s2.indexOf("/")));
   int day2 = Integer.parseInt (s2.substring (s2.indexOf("/")+1));
   return month1 < month2 || (month1 == month2 && day1 < day2);
}
```

Note that precedes is defined *outside* the IslamicDate class, so the private variables within IslamicDate are inaccessible. If the month and day components of each date are known, the computation is easy; thus access to the private IslamicDate variables significantly simplifies the problem. We deducted 4 points for this error, which was *very* common. (An exception was a solution that used the private variables in the context of a loop or recursion structured as described above. This earned only a 2-point deduction.)

A similar error, also earning a 4-point deduction, was to use one of the daySpan methods, which do not appear in the abbreviated IslamicDate class.

Since the precedes method is to be called from DateTester.main without any DateTester object having been initialized, it must be declared as static. 1 point was deducted for omitting the static modifier, or for any other error in the method header. (Omitting the IslamicDate declarations from the parameter list, i.e. saying

```
public static boolean precedes (d1, d2)
```

was a somewhat less common way to lose this point.) We did not deduct any points for declaring the method as private rather than public.

There were numerous opportunities for logic errors. An off-by-one error generally lost 1 point. Off-by-one errors included both errors in data comparisons and errors in String processing for those who used the toString approach. It also included solutions that returned true if the dates were equal. Most other minor logic errors, e.g. using an incorrect comparison operator or nesting ifs incorrectly, also lost 1 point.

A more serious logic error was to somehow compute the number of days between the two dates, without checking whether one has wrapped around past the end of the year when computing this difference. This often involved a comparison with the total number of days in a year, but with no comparison to the first or last day of the year. This earned a 2-point deduction. Even more serious was to handle only consecutive dates, for example by calling d1.tomorrow ( ) and comparing the resulting date to d2 without a loop or recursion. This error lost 5 points.

Other errors, all worth a 1-point deduction, included the following:

- Side effects, i.e., calling makeTomorrow on one of the input objects. (A subtle case was modifying a temporary date variable that was sharing one of the input objects.) The specification says that nothing should be modified.
- Use of = instead of == or use of .equals on int (primitives).
- Returning “true” or “false” strings rather than the boolean values true or false.
- Various syntax errors such as using Scheme syntax or omitting braces.

Problem 3 (5 points)

This problem involved analyzing code similar to that from lab assignment 3. It was problem 2 on versions C and D. Version A’s method was

```
public boolean contains1MoreThan (String s) {
   if (myString.length( )==0) {
      return false;
   } else if (s.length( )==0) {
      return true;
   } else {
      StringToCheck5 remainder
         = new StringToCheck5 (myString.substring(1));
      if (myString.charAt(0) == s.charAt(0)) {
         return remainder.contains1MoreThan (s.substring(1));
      } else {
         return remainder.contains1MoreThan (s);
      }
   }
}
```

Version B's code was the same, except that the order of the first two comparisons was switched:

```
if (s.length( )==0) {
   return true;
} else if (myString.length( )==0) {
   return false;
} else { ...
```

Part a, worth 2 points, was to determine if the method would crash. You earned 1 point for the answer and the other for the explanation. The answer we expected was "no", since the length comparisons guarded the subsequent uses of charAt and substring. Since grading the exam, however, we noticed that the method precondition did not rule out a value of null for myString. We will award 1 or 2 points for the answer "yes", provided that your explanation mentions the possibility of myString containing null. (Whether you get 1 or 2 points depends on the rest of your answer for that part.)

A common error on part a was to claim that a crash would result from myString or s containing exactly one character. Note, however, that charAt(0) is the String equivalent of Scheme's car function; similarly, the String equivalent of cdr is substring(1). Applying substring(1) to a one-character string gives the empty string and produces no error. (StringToCheck2.contains1MoreThan from lab assignment 3 involves the same operations, and doesn't crash.) The erroneous claim may have resulted from confusing the empty string with a null String reference.

Questions for part b differed in the two versions. On version A, you were to describe all pairs of Strings myString and s for which contains1MoreThan should return true but doesn't. On version B, we wanted just the opposite: a description of all pairs of Strings myString and s for which contains1MoreThan should return false but doesn't. Here's the analysis a correct answer might have involved.

- Examining the base cases, we see that version A's method returns false given any empty string myString. If myString is initially empty, that's the correct result, since an empty string can't possibly be the result of inserting a single character into anything. We thus must examine the recursive cases to see if somehow myString can run out at the same time as or before s. There are two possibilities: either the first characters match, so the remainders of the two strings are compared, or they don't match, so the rest of myString is compared to s. If both remainders are compared, myString's length relative to s's stays the same. Thus we consider the simplest alternative, namely the case where myString shrinks but s doesn't, yet myString is still the result of inserting a single character into s. This happens for the strings "ab" and "b". More generally, it happens for any myString that results from adding a single character in a position other than the end of s. Here are some examples:

myString	s	*explanation*
xabc	abc	returns false, should return true
axbc	abc	returns false, should return true
abxc	abc	returns false, should return true
abcx	abc	returns true, correctly
abcc	abc	returns true, correctly

- In version B, we similarly note that contains1MoreThan returns true if s is empty, regardless of the length of myString. Thus when myString contains zero or more than one character, the returned value is incorrect. Now we examine the recursive cases to see how this base case might be reached. There are two possibilities: either the first characters match, so the remainders of the two strings are compared, or they don't match, so the rest of myString is compared to s. More generally, the length of s decreases each time a matching pair of characters in the two strings is detected. Thus whenever every character in s has a matching character in myString—that is, all the characters in s occur in myString, in the same sequence—contains1MoreThan will return true. The return value is incorrect when either myString and s are identical or when myString is the result of inserting two or more characters into s. Here are some examples:

myString	s	*explanation*
xyabc	abc	returns true, should return false
axbyc	abc	returns true, should return false
abcxy	abc	returns true, should return false
abc	abc	returns true, should return false

Part b, worth 3 points, was graded as follows. You earned 1 point for a correct example, provided that you also didn't provide any incorrect examples. You earned 2 points for specifying an infinite set of strings that was a subset of the correct set; an example 2-point answer, in version B, was to say that only equal strings cause false positives. In version A, saying all pairs myString and s where myString is the result of appending a character to s—exactly the opposite of what we intended—earned 1 point out of 3.

Problem 4 (4 points)

This problem was based on lab assignment 4. Version A was identical to version B except that the parts were reversed. It was problem 1 on versions C and D.

Here is an Account constructor that throws IllegalArgumentException if its argument is negative. (The assumption is that it would *replace* the 1-argument constructor currently provided in the class.)

```
public Account (int balance) {
   if (balance < 0) {
      throw new IllegalArgumentException ("Balance < 0:" + balance);
   }
   myBalance = balance;
   myParent = null;
}
```

IllegalArgumentException is an "unchecked" exception, so you don't need throws IllegalArgumentException in the header; it doesn't hurt to include it, though. The assignment statements may come either before or after the balance test. You don't need the assignment to myParent since it would be set to null by default.

1 point was deducted from the constructor solution for each of the following errors; both were missing some important part of the exception throw.

- missing new;
- missing argument to the IllegalArgumentException constructor.

Bad logic (usually throwing the exception for a *positive* balance) also received a 1-point deduction. Solutions that had all the necessary parts received a $\frac{1}{2}$-point deduction for incorrectly specified parts, for example:

- throwing the wrong type of exception;
- saying "throw" for "throws" or vice-versa (on some exams, 1 point was mistakenly deducted for this error);
- adding throws IllegalArgumentException before the method argument list rather than after.

The other part of this problem involved testing a version of the 1-argument constructor written by someone else: "initialize an Account with a negative balance and then print a suitable message about what happened". A full-credit solution printed *some* message regardless of whether or not the constructor threw the exception. Here's a solution.

```
public static void main (String [ ] args) {
   try
      Account test = new Account (-3);
      System.out.println ("Account initialized with negative balance!");
   } catch (Exception e) {
      System.out.println ("Exception correctly thrown "
         + "for account with negative balance.")
   }
}
```

A frequent error was the failure to include a call to println immediately after the constructor call, to print an error message if the constructor *did not* throw an exception. This lost 1 point. (This error occurred so often that some graders resorted to shorthand: "DPSM" means "Didn't Print Suitable Message".) Other 1-point errors were access of Account private variables—the main program may be in a "Tester" class rather than in the Account class—and method calls with incorrect syntax, e.g. using acct.balance instead of acct.balance ( ).