Spring, 1992 Name: ________________________________

CS 541 — Programming Languages 1

Object-Oriented Programming Test

(Problem Solving, Version 1)

Before you begin, please take a moment to look over the entire test so that you can budget your time.
If you get an incomplete on this test, you can get at most a 'B' on the makeup test.
For programming problems, there is usually more than one right answer, hence it is important to present with your program evidence that it is correct. Answers to programming questions will be scored on the following points:

- elegance and clarity (conciseness, using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution, and how clear and convincing your argument is that the code is correct)
- correctness (whether your program solves the given problem correctly in the given programming language)

You should try to eliminate all syntax errors from your programs, as they will affect both the correctness and clarity of your solution; you may receive no points for programs with major syntax errors. Run-time space and time efficiency of programs is not important, unless the problem says so. Also, checking of external inputs, prompting, labeling output, and other conventions for dealing with humans are not important (for this class), unless the problem says so. The breakdown of points between these categories is roughly as follows.

- 60% for elegance and clarity (using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution),
- 40% for correctness (whether your program solves the given problem correctly in the given subset of the given language)

You are encouraged to use auxiliary subroutines whenever they are helpful.
For this test, you may use anything documented in the Overview and Specification of the built-in types in Little Smalltalk. If you wish to use functions from your solutions in other parts of the test, you must clearly state where (which problem) we can find those function definitions.
You may write code for any class, including new classes you may create. However, you may not access the instance variables of classes you do not create. (That is, you should use all built-in classes as a client.)
When you write code for classes, give both the message send that creates the class and the code of the methods. Make it clear by indentation where methods start and stop. Make it clear which methods go with which classes.

1. (5 points) Write a method `collatz` for class `Integer` such that for an integer n, n `collatz` returns $(3n+1)/2$, if $n \equiv 1 \pmod 2$, and $n/2$ if $n \equiv 0 \pmod 2$. That is, for $n > 0$, n `collatz` takes odd n to $(3n+1)/2$ and even n to $n/2$.

2. (15 points) Write a method, `hasFirst:then:` for class `List` such that

   ```
   lst hasFirst: x then: y
   ```

   is `true` if somewhere in the List `lst`, first the object `x` appears, followed consecutively by `y`. For example, if the global variable `y5126` is defined by

   ```
   y5126 <- ((((List new) addLast: 5) addLast: 1) addLast: 2) addLast: 6
   ```

   then `y5126 hasFirst: 1 then: 2` should return `true`, but `y5126 hasFirst: 1 then: 6` should return `false`. The method `hasFirst:then:` should not modify any of its arguments (If you need to, make a copy and then modify the copy.) Use `==` to compare elements of the list against the arguments `x` and `y`.

3. (30 points) Design and implement a class `BinTree` of binary trees as a subclass of `Collection`. The class `BinTree` should have at least six (6) operations that it does not inherit from its superclass. You should also define the `size` and `do:` methods so that a `BinTree` can act like a `Collection` (see sections 6.1.1 and 6.1.2 in the Little ST specifications document). Be sure that you explain in some way other than code what your code is supposed to do.