Your name _____

Check here if you used a reference sheet ___

If so, please submit the sheet along with your exam solutions.

☐

☐

Harvey Mudd College

CS 60 Mid-Term Exam

Fall semester, 2002

Six Problems

100 Points

Closed book

Instructions

During the exam, the only item to which you may refer is your own reference sheet, one double-sided sheet of 8.5" x 11" paper.

The exam has a time limit of 1 hour 15 minutes. Work so as to maximize total points; do not get stuck on one problem at the expense of others. It is suggested that you look over the exam first to get an idea of how to apportion your time.

Please provide answers to the problems directly on these pages.

For each problem, the action items are shown in boldface.

When code is requested, exhibiting the correct idea is more important than syntactic precision. However, keep your definitions clean and readable. Use the clearest program structure possible.

1. [20 points]

The rex language includes a built-in function `count` defined as follows:

```
count(P, L)
```

where `P` is a predicate and `L` is a list, returns the number of items in `L` that satisfy `P`. For example,

```
count(odd, [1, 2, 3, 4, 5]) ==> 3
```

a. [**3**/20 points] Suppose that `count` were not available, but functions `keep` and `length` were.
Define `count` in terms of these other functions.

b. [**5**/20 points] Suppose that no high-level functions were available.
Define `count` using recursion.
Bonus: You get 5 extra points if you give a solution that is tail-recursive.

☐

☐

c. [**5**/20 points] The higher-order function `makeCounter` has a predicate as its single argument. It returns a function that also has one argument, a list. The latter

function, when applied to a list, returns the number of elements for which P is true. For example, if we have the definition

```
counter = makeCounter(odd);
```

then a use of this counter might be

```
counter([1, 2, 3, 4, 5]) ==> 3
```

Construct a rex definition of `makeCounter` (you may use your work from previous parts):

d. [**7**/20 points] We now want to redefine `count` to apply to *unlabeled trees* represented as lists. We want to count the number of leaves in the tree satisfying the predicate. For example:

```
count(odd, [[0, 1], [2, 3], [4, [5, 6, 7]]]) ==> 4
```

Define the tree version of `count`. You may use recursion or higher order functions, as you see fit. (The predicate `atomic` differentiates non-lists from lists.)

☐

☐

☐

☐

☐

☐

2. [20 points]

We wish to implement `count` mentioned in problem 1, parts a and b for OpenLists in Java.

a. [**5**/20 points] **Give the two main reasons why the following implementation is undesirable.**

```
static int count(Function1 P, OpenList L)
  {
  int result = 0;

  for( int i = 0; i < L.length(); i++ )
    {
    if( ((Boolean)P.apply(L.nth(i))).booleanValue() )
      {
      result++;
      }
    L = L.rest();
    }
  return result;
  }
```

b. [**5**/20 points] **Give an implementation that corrects the undesirable aspects in the preceding implementation.**

□
□
□
□

c. [**10**/20 points] **Modify your implementation in part b. so that it works for the tree case, as in problem 1d.**

□
□

3. [**20** points]

Recall that the Java interface `Iterator` has the following methods:

```
Object next();
```

```
boolean hasNext();
```

(It also has method `remove()`, but ignore that for this problem.)

Code a class `OpenListIterator` that implements `Iterator` for our `OpenList` class. An object of this class will return the elements of the list, beginning with the first, until there is none left. This implementation will thus allow us to apply any code using `Iterator` to `OpenLists`. The class's constructor will have as its only argument the `OpenList` over which iteration is to occur.

□

□

4. [**10** points]

Consider the following Java class definition in the context of closed lists:

□

```
Class Cell
{
Object data;
Cell next;

Cell(Object data, Cell next)
{
this.data = data;
this.next = next;
}
}
```

□

Suppose that variables `ap` and `bp` have been initialized *as if* by the following statements:

□

```
Cell ap = new Cell("a", new Cell("c", null));
```

```
Cell bp = new Cell("b", null);
```

□

Give the sequence of assignment statements that will insert the cell containing "b" into the list destructively, between the existing cells containing "a" and "c". (Don't create any new cells.) **Draw the corresponding picture first.**

□

□

□

□

□

□

□

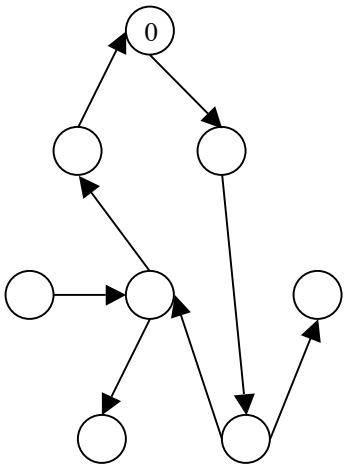
□

□

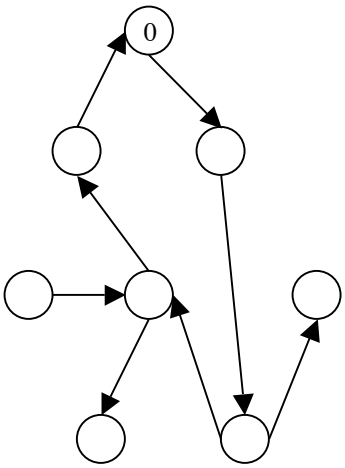
5. [**10** points]

On the two identical copies of the directed graph below, **number the nodes in the left graph in an order that could occur in a depth-first search of the graph. Number the nodes in the right graph in an order that could occur in a breadth-first search of the graph**. Search from the node labeled 0 in both cases. (Nodes not reachable from that node don't get numbered.)

□Depth-First:



Breadth-First:



6. [**20** points]

An example of exploiting inheritance concerns shapes in graphical programs. Various shapes, such as `Oval`, and `Rectangle`, are to be manipulated on the screen in a drawing program. In addition, it is possible to have a `Group` shape, which is a collection of constituent shapes that move as a unit. The following additional design points are of importance:

> Certain methods in our application should be able to accept a shape as an argument regardless of which kind of shape it is.
>
> Every shape has, at a minimum, an `x` and `y` position representing the upper left-hand corner of an imaginary box bounding the shape. Similarly, every shape has a `width` and `height`, defined to be the corresponding parameters of this box.
>
> Every shape has a `move` method that will set its `x` and `y` to a designated position and a `draw` method that will cause it to be drawn on some `Graphics` object. The draw method consults the `x` and `y` of the shape to determine where to draw it.

Show appropriate class definitions for a design that exploits inheritance. Give key instance variables in appropriate places. Avoid replicating methods in more than one class unnecessarily. Use `OpenList` to record the shapes in a Group. **You do not need to code the methods**, but show headers for the methods of interest to an application program that uses your implementation.