Instructions

1. This is an open-notes, open-book, open-computer, open-Internet, take-home exam.
2. **You have 5 hours to work on this exam.**
3. **You may also take one contiguous 15-minute break during the exam, and that time will not count towards the 5 hours.**
4. **Except for the 15-minute break, this exam has to be taken in one sitting.**
5. Please make sure you have all 10 pages of this exam.
6. This exam is comprehensive.
7. You will not be penalized on trivial syntax errors, such as a missing parenthesis. Multiple errors or errors that lead to ambiguous code will have points deducted, however.
8. In all of the questions, feel free to write additional helper methods to get the job done.
9. The emphasis is on correctness of the code, not efficiency or on simply generating the right result.
10. You are free to use any code that was given to you in the lectures and labs.
11. Code should be generic whenever possible.
12. **When you are done, create a zip file of all your source code and this edited exam, and submit it on OWL-Space by the end of the exam period (Wednesday, December 17, 2008, 11:59 PM).**
13. Do not discuss this exam with anyone other than the instructors or the teaching assistant until after the exam period is over.

Please write and sign the Rice Honor Pledge here:

1	**2a**	**2b**	**3a**	**3b**
/20 pts	/30 pts	/5 pts	/40 pts	/3 pts

4a	**4b**	**4c**	**4d**	**4e**	/100 pts
/3 pts	/2 pts	/3 pts	/2 pts	/2 pts	

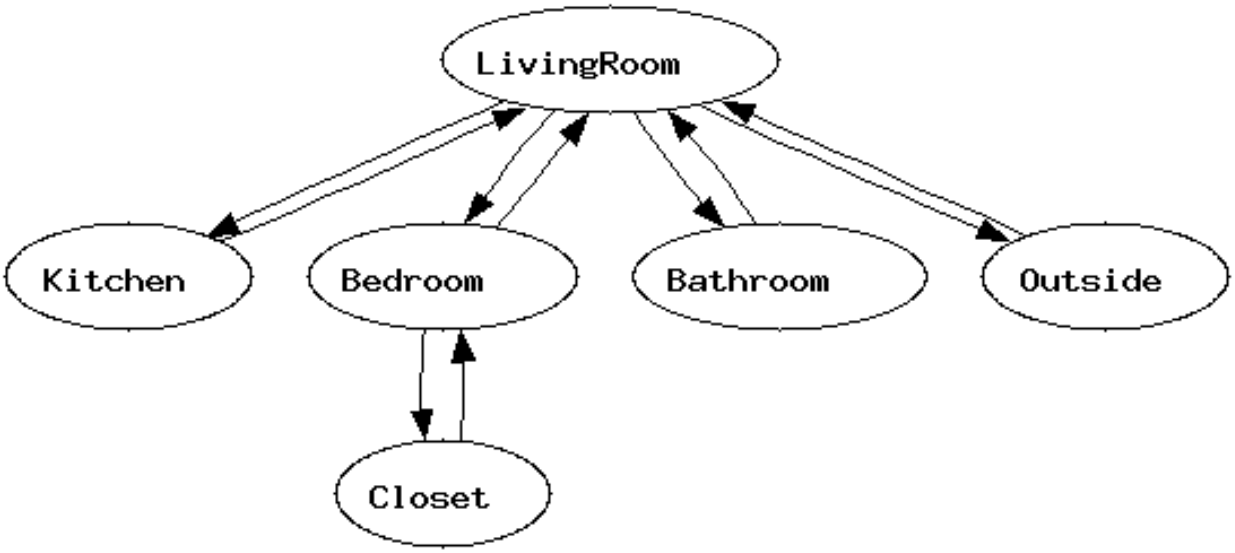
(10% are extra credit)

1. **Parsing (20 pts)**:

You are to write a parser for a language describing a directed graph.

A *graph* is a set of *nodes*. Each node has a *name* that uniquely identifies the node in the graph, i.e. there may be only one node with a given name in the graph. Each node also has a set of other nodes that are connected to the node through an *edge*, i.e. a one-headed arrow.

The graph below has six nodes: LivingRoom, Kitchen, Bedroom, Bathroom, Outside and Closet. The arrows indicate the edges between the nodes.



Let the following grammar define a language to define graphs:

G = (N, T, P, S) with
N = { G, H, N, F } (non-terminals)
T = { IdToken, ;. → } (terminals; IdToken, semicolon and arrow, consisting of a dash and greater-than ->)
P = { G ::= H | Empty
H ::= N ; G
N ::= IdToken -> E
E ::= F | Empty
F ::= IdToken E } (production rules)
S = G (start symbol)

Then the graph above is described by the following string:

```
LivingRoom -> Kitchen Bedroom Bathroom Outside;
Outside -> LivingRoom;
Kitchen -> LivingRoom;
Bedroom -> LivingRoom Closet;
Closet -> Bedroom;
Bathroom -> LivingRoom;
```

Note that a graph may be completely empty, and that it is legal to have no IdTokens to the right of the arrow. That means that the following string is legal:


```
JustOneNode -> ;
```

Also note that a node may have itself as neighbor, in which case the edge loops back to the node itself:


```
JustOneNode -> JustOneNode;
```

Write the code to parse the grammar above using `SequenceFact`, `MultiSequenceFact`, `CombinationFact`, `TerminalSymbolFact`, `MTSymbolFact`, and/or `ProxyFact`.

Please complete the `graph/src/GraphFrame.java` file in the accompanying zip archive. The parts you need to edit are marked with `TODO: 1`

2. **Model Conversion**:

a. (30 pts) **Write an extended visitor `ToModelAlgo` that takes the parse tree from the parser above and creates a model consisting of a `Graph` object and several `Node` objects.**

The code for this model can be found in the `graph/src/model/Graph.java` and `graph/src/model/Node.java` files.

Complete the `ToModelAlgo` visitor so that it converts the tree generated by the parser into objects of this more convenient model.

Please complete the `graph/src/parser/visitor/ToModelAlgo.java` file in the accompanying zip archive. The parts you need to edit are marked with `TODO: 2a`

b. (5 pts) **Add code to the `graph/src/GraphFrame.java` file so that the result of the parser (variable `result`) is converted into a `Graph` (field `_g`). Then append `_g.toString()` to the `StringBuilder sb`.**

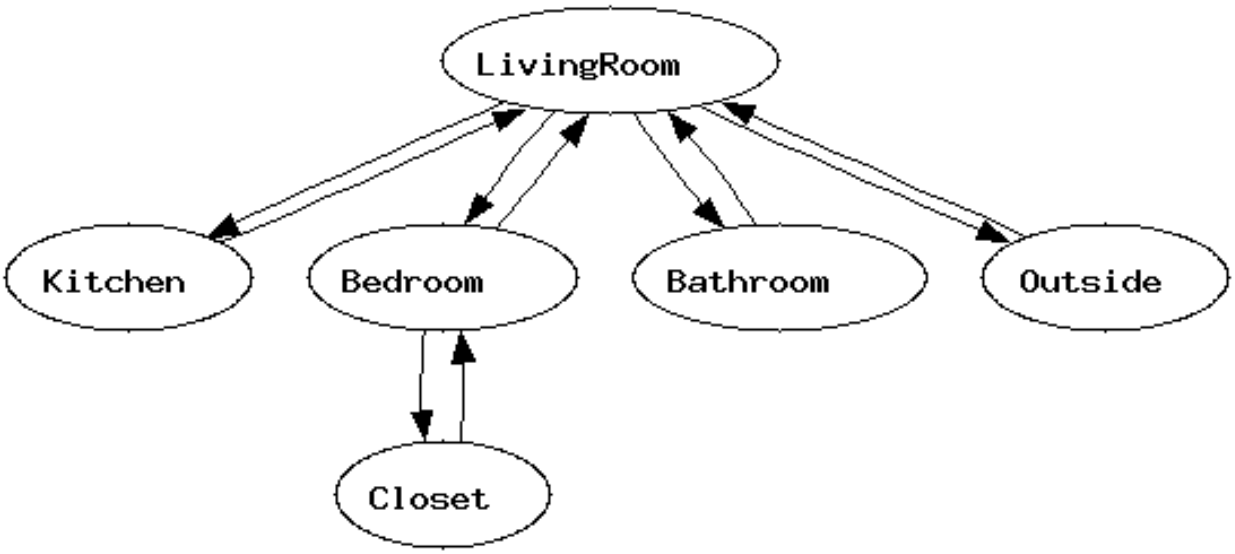
Please complete the `graph/src/GraphFrame.java` file in the accompanying zip archive. The parts you need to edit are marked with `TODO: 2b`

3. **Graph Search**:

a. (40 pts) **Write a visitor on graph nodes from the model above that can find a path between two nodes.**

The visitor, which uses the extended visitor framework, can be found as nested class `SearchVisitor` inside `graph/src/model/GraphSearch.java`. The surrounding class `GraphSearch` provides a convenience `method public IList<Node> search(Graph g, Node start, final Node end)` and also stores the list and RAC (restricted access container) factories to be used. When `search` is called, it will return a list that contains the path of nodes from the start node to the end node. If no path exists, an empty list will be returned.

Example:



`search(g, Outside, Closet)` returns the list with the nodes **(Outside, LivingRoom, Bedroom, Closet)**
`search(g, LivingRoom, LivingRoom)` returns the list with the node **(LivingRoom)**

Example:

`search(g, CantGetThere, FromHere)` returns the empty list **()**

During a graph search, the nodes to visit next are stored in a RAC (i.e. a stack or a queue, or potentially a priority queue), similar to the breadth-first traversal of a tree.

The `SearchVisitor` extends `GraphVisitor<IList<Node>,Void,IList<Node>>` , which means that it returns an `IList<Node>`, it does not use varargs parameters (they are `Void`). The third type parameter, `IList<Node>`, denotes the command for each node has a state of type `IList<Node>` associated with it. **The state records the path back to the start node from when the node was first reached (i.e. if it is reached again, the path will not be updated).**

There are two visitor commands, `Visited` and `Unvisited`, both of which extend `NodeCommand<IList<Node>,Void,IList<Node>>`, which makes sense, since the visitor has the same type arguments: The commands return `IList<Node>`, do not use varargs parameters, and they have a "state" of type `IList<Node>` associated with them.

The `GraphVisitor<R,P,S>` extends `AExtVisitor<R,String,P,Node>`; it sets the index type to `String` and the host type to `Node` for convenience reasons. In addition, it uses the type parameter `S` to define the type of the state that the commands have. `NodeCommand<R,P,S>` extends `IExtVisitorCmd<R,String,P,Node>`; it also sets the index type to String and host type to `Node` and uses `S` for the state

For each node, these commands tell the search algorithm what to do with a node when it is retrieved from the RAC. Initially, the commands for all nodes are set to `Unvisited` with an empty list, meaning they have never been reached yet and no path there has been found. We will make use of the fact that the commands are stored in a hash table and can be updated dynamically: The behavior of the visitor for a node changes after the node has been visited; if the visitor reaches the nodeagain, it will act differently.

The graph search algorithm simply executes the command for the current node, whether it is `Visited` or `Unvisited`, and the commands make sure that the right thing happens. We therefore need to define those commands.

Unvisited:
The current node has just been reached for the first time.
1. Append the current node (host) to the state.
2. Set the command for the current node to Visited with the updated state.
3. Check if the current node is the end node.
   - If it is, return the updated state (i.e. return the path from the end node back to the start node).
4. For all neighbors of the current node, do…
   1. Get the command for the neighbor's node from the visitor's hash table.
   2. Set the state of the neighbor's command to the path of the current node (the host).
   3. Put the updated command for the neighbor's node back in the hash table.
   4. Add the neighboring node to the RAC
5. Process the next node from the RAC (done in `processNext()`, see below) and return whatever it returns.

Visited:
The current node has been reached before. We don't want to do any additional processing for this node; therefore, we just…
1. Process the next node from the RAC (done in `processNext()`, see below) and return whatever it returns.

`processNext()`:
The purpose of this method, called from the `apply` methods of both the `Unvisited` and `Visited` commands, is to take the next node from the RAC and continue the search there. Of course, it has to deal with the possibility that the RAC may be empty, meaning all nodes in the graph that can be reached have been processed already.
1. Execute a visitor on the RAC to determine if it is empty or non-empty.
   a. If it is empty, that means there are no more nodes to process, and no path from the start node to the end node has been found. Return the empty list **()**.
   b. If it is non-empty, there are more nodes to process.
      1. Remove the next node from the RAC.
      2. Execute the `SearchVisitor` on it (`SearchVisitor.this`).
      3. The return value from the `SearchVisitor` is a list with nodes. It is empty if no path was found (see 1.a. above), or contains the path from the end node back to the start node (see Unvisited: 3.). To find out which of these cases it is, run a visitor on the return value from the `SearchVisitor`.
         a. If the return value is the empty list, process the next node from the RAC by calling `processNext()` and return whatever it returns.
         b. If the return value is non-empty, then that is the path back from the end node to the start node. Return it!

Depending on the kind of RAC, the order in which nodes are visited can differ: With a queue, the search algorithm will first visit the start node, then all the nodes one edge away from the start node, then all the nodes two edges away, and so on. This is a breadth-first approach. With a stack, the search algorithm will go as deep into the graph as possible, using a depth-first approach. The order also depends on the order the nodes are stored in the sets of neighbors.

Below is a simulation of the search from Outside to Closet in the big graph. We need to keep track of what is in the RAC, what commands are set for the different nodes, and what state is associated with the commands (i.e. the path). Assume the RAC we use is a queue.

Initial state:
RAC = ()
Outside = Unvisited ()
LivingRoom = Unvisited ()
Kitchen = Unvisited ()
Bathroom = Unvisited ()
Bedroom = Unvisited ()
Closet = Unvisited ()

Execute visitor on Outside.
Outside's neighbors added to RAC: LivingRoom

RAC = (LivingRoom)
Outside = Visited (Outside)
LivingRoom = Unvisited (Outside)
Kitchen = Unvisited ()
Bathroom = Unvisited ()
Bedroom = Unvisited ()
Closet = Unvisited ()

Execute visitor on LivingRoom.
LivingRoom's neighbors added to RAC: Outside, Kitchen, Bathroom, Bedroom

RAC = (Outside, Kitchen, Bathroom, Bedroom)
Outside = Visited (Outside)
LivingRoom = Visited (LivingRoom, Outside)
Kitchen = Unvisited (LivingRoom, Outside)
Bathroom = Unvisited (LivingRoom, Outside)
Bedroom = Unvisited (LivingRoom, Outside)
Closet = Unvisited ()

Execute visitor on Outside.
Outside has already been visited so process the next node.
RAC = (Kitchen, Bathroom, Bedroom)

Execute visitor on Kitchen.
Kitchen's neighbors added to RAC: LivingRoom

RAC = (Bathroom, Bedroom, LivingRoom)
Outside = Visited (Outside)
LivingRoom = Visited (LivingRoom, Outside)
Kitchen = Visited (Kitchen, LivingRoom, Outside)
Bathroom = Unvisited (LivingRoom, Outside)
Bedroom = Unvisited (LivingRoom, Outside)
Closet = Unvisited ()

Execute visitor on Bathroom.
Bathroom's neighbors added to RAC: LivingRoom

RAC = (Bedroom, LivingRoom, LivingRoom)
Outside = Visited (Outside)
LivingRoom = Visited (LivingRoom, Outside)
Kitchen = Visited (Kitchen, LivingRoom, Outside)
Bathroom = Visited (Bathroom, LivingRoom, Outside)
Bedroom = Unvisited (LivingRoom, Outside)
Closet = Unvisited ()

Execute visitor on Bedroom.
Bedroom's neighbors added to RAC: LivingRoom, Closet

RAC = (LivingRoom, LivingRoom, LivingRoom, Closet)
Outside = Visited (Outside)
LivingRoom = Visited (LivingRoom, Outside)
Kitchen = Visited (Kitchen, LivingRoom, Outside)
Bathroom = Visited (Bathroom, LivingRoom, Outside)
Bedroom = Visited (Bedroom, LivingRoom, Outside)
Closet = Unvisited ()

Execute visitor on LivingRoom.
LivingRoom has already been visited so process the next node.
RAC = (LivingRoom, LivingRoom, Closet)

Execute visitor on LivingRoom.
LivingRoom has already been visited so process the next node.
RAC = (LivingRoom, Closet)

Execute visitor on LivingRoom.
LivingRoom has already been visited so process the next node.
RAC = (Closet)

Execute visitor on Closet.
Closet is the end node! Return updated state of Closet:

RAC = (LivingRoom, LivingRoom, LivingRoom, Closet)
Outside = Visited (Outside)
LivingRoom = Visited (LivingRoom, Outside)
Kitchen = Visited (Kitchen, LivingRoom, Outside)
Bathroom = Visited (Bathroom, LivingRoom, Outside)
Bedroom = Visited (Bedroom, LivingRoom, Outside)
Closet = Visited (Closet, Bedroom, LivingRoom, Outside)

The list (Closet, Bedroom, LivingRoom, Outside) is returned by the visitor at this point. The `search` method then reverses this list to yield (Outside, LivingRoom, Bedroom, Closet).

Please complete the `graph/src/model/GraphSearch.java` file in the accompanying zip archive. The parts you need to edit are marked with `TODO: 3a`

You should now have a complete `GraphApp` program that can parse graph definitions, let you select start and end nodes, and then search for a path using breadth-first and depth-first search.

Note: The graph search algorithm is just a visitor in the extended visitor framework. The nodes are the hosts. What's different is that the commands for the different hosts can change dynamically!

Note: If you did not get problems 1. and 2. to work, you can press the "Test" button to use a hard-coded graph.

b. (3 pts) **The state in the commands stores a list representing the path from the node associated with the command back to the start node. So if we want the path from the start node to that node, we have to reverse the list first. Why do we have to reverse the list in the method? Why can't we just search from the end note to the start node?**

4. **Sorting: Short Answer Questions**

Consider the set with the elements {2, 4, 6, 7, 8, 10, 9, 7, 5, 3, 1}.
A **permutation** of that list is a list that contains each of the elements in the set above exactly once – the order does not matter.

Example: Both (1, 10, 2, 9, 3, 8, 4, 7, 5, 6) and (2, 4, 6, 7, 8, 10, 9, 7, 5, 3, 1) are permutations of the set above.

Let all sort algorithms sort in an ascending manner (from least to greatest, 1, 2, 3…).

a) (3 pts) **Provide a "lucky" permutation of the set above that allows Insertion Sort to execute as in a best-case scenario.**

b) (2 pts) **In big-oh notation, what is the cost of Insertion Sort in that best-case scenario above?**

c) (3 pts) **Provide two pathological ("unlucky") permutation of the set above that make QuickSort execute in a worst-case scenario. Assume that the first element in the array is chosen as pivot element.**

d) (2 pts) **In big-oh notation, what is the cost of QuickSort in that worst-case scenario above?**

e) (2 pts) **Suggest a way to improve QuickSort to make such a worst-case scenario less likely.**