COP5555 Fall 2013
Assignment 4

Assigned Oct 18
Due: Wed Oct 30 at 11am

1. Implement a set of ML functions that together, implement a FIFO queue where the queue is represented as a list. The names and signatures of the functions are given. Fill in the implementation of the function. You will need to define an exception called Empty.

```
returns true if the queue is empty, false otherwise
val isEmpty = fn : 'a list -> bool

takes a queue and an element and returns a new queue containing the element
val enqueue = fn : 'a list -> 'a -> 'a list

returns the first element in the queue, throws an Empty exception if applied to an
empty queue
val first = fn : 'a list -> 'a

returns the list after the first element has been removed , throws an Empty exception if
applied to an empty queue
val dequeue = fn : 'a list -> 'a list
```

   Turn in a text file called Q1.sml containing all of your functions in a form that will work when loaded with

   use “Q1.sml”;

2. A more efficient implementation of a FIFO queue uses two lists, one where items are inserted and one where items are removed. All of the items in the insertion list are moved to the removal list when dequeue is evaluated with a two-list queue when the removal list is empty while the insertion list is not.
   a. Give an implementation of this queue with the following functions. The meaning is the same as above. You may want to write additional helper functions.

```
val isEmpty = fn : 'a list * 'b list -> bool
val enqueue = fn : 'a list * 'b -> 'a -> 'a list * 'b
val first = fn : 'a list * 'a list -> 'a
val dequeue = fn : 'a list * 'a list -> 'a list * 'a list
```

b. Explain why the two-list implementation is more efficient than the implementation of part a.

Turn in a text file called Q2.sml containing all of your functions in a form that will work when loaded with

use "Q2.sml";

Put your answer to part b in the same file as a comment (delineated with (*.. *)). Make sure that this doesn't cause errors.

3. Implement type checking for our language as specified in the attached document. It should implement the ASTVisitor interface and have the following elements in addition to the methods and members you need to add to implement type checking. Note the package.

```
package cop5555fa13.ast;
public class TypeCheckVisitor implements ASTVisitor {


	public TypeCheckVisitor(){
		symbolTable = new HashMap<String, SymbolTableEntry>();
		errorNodeList = new ArrayList();
		errorLog = new StringBuilder();
	}

	public List getErrorNodeList(){return errorNodeList;}
	public boolean isCorrect(){
		return errorNodeList.size()==0;
	}
	public String getLog(){
		return errorLog.toString();
	}


	List<ASTNode> errorNodeList;
	StringBuilder errorLog;


	Everything else you need to implement the ASTVisitor and perform type checking.

}
```

When an error is discovered, the node you are currently checking (i.e. if an error is detected in method visitX, the current X node should be added to the error NodeList), and a useful error message should be appended to the errorLog. As usual, we will not grade the contents of your error messages, although giving useful messages is highly recommended. After an error is detected, you may as well recover and continue type checking, but to save Nakul's sanity, the grading script will only check the first error. Whatever you decide to do, make sure that the node for at least the first error is correctly placed in the errorNodeList and that your program doesn't crash.

Assuming that prog is an AST of a correct program from your Parser, the following code fragment should result in type checking.

```
TypeCheckVisitor v = new TypeCheckVisitor();
prog.visit(v, null);
if (!v.isCorrect()){
      System.out.println(v.getLog());
      //we're done with this program, exit or do something else sensible
}
else //program is correct
     go on to code gen (in assignments 5 and 6)
```

Submit three files to e-learning:

Two separate text files containing your responses to questions 1 and 2.

A jar file called Assignment3.jar containing your implementation of Scanner.java, Parser.java and any other classes you might have added. You do not need to turn in your test cases. **We will recompile and call the public methods from our test program. Do not change the signatures of the public methods or the package declarations or you will break the test script.**

Suggestion: if you weren't in class, make sure to watch the beginning of the 18Oct lecture before starting the project.