Assignment 2

due 27/2/2014

1. This assignment involves the implementation of an interpreter for a simple low-level programming language known as three-address code (TAC).

2. Download the file `a2.tgz` and untar it. This contains some materials you will need for the assignment. You should work within this directory.

3. Submit your work (as an `a2.tgz` file) using the Moodle sever on or before the date indicated above.

Our Language

4. Three-address code is a simple, low-level programming notation that is based on the following limited "instruction set". The key instruction is that of simple assignment statement comprising at most three variables/values (one on the LHS, at most two on the RHS)– hence the name.

```
                          # This is a comment!

x := y op z;              # primitive assignment; two vars/values on RHS only
                          # op in {+, -, *, /, ==, !=, <, <=, >, >=}
x := y;                   # copy assignment
read x;                   # prompt user to enter (integer) value, place value in x
write x;                  # write value of x (integer) to the screen
if ( x == 0 ) goto l;     # jump to l if x is false  (zero)
if (x != 0 ) goto l;      # jump to l if x is true (nonzero)
goto l;                   # jump unconditionally to l
l:                        # jump target l
halt;                     # terminate program execution
```

Note that: (1) Instruction semantics is as you would expect based on their counterparts in *C*; (2) Instructions are terminated by semicolons (apart from the label instruction); (3) Programs manipulate integer quantities only; (4) There are no variable declarations; (5) Identifiers and labels are alphanumeric strings, beginning with a letter and are case sensitive; (6) The symbol := denotes assignment; (7) Comments begin with # and extend to end of line.

5. Appended at the end of this sheet is a sample TAC program (`fact.tac`) that takes in a value x and that calculates and prints $x!$ (factorial). The comments on the right indicate "high-level" pseudocode to indicate the program's structure.

Our Interpreter

6. A range of classes provided. You will need to modify only `TacExecutionEngine.java`.

   - Various ADT classes: ArrayBasedMap etc. that you may find useful in developing your code.

- Various TAC programs: In the a2 directory, you will find a range of `.tac` programs, which will be useful when debugging your code.
- `TacScanner2` (complete): Reads the contents of a textfile and creates a TAC-program object from the contents.
- `TacInstruction2` and `TacProgram2` (both complete): Objects of the former class represent individual TAC instructions, while the latter represents a complete TAC program.
- `TacExecutionEngine`: A stub is provided, but you will need to complete this class.

Our Goal

7. Complete `TacExecutionEngine.java` based on the stub provided.
8. The constructor of `TacExecutionEngine` takes a TAC program (`TacProgram2`) as an argument; method `execute` simulates the execution of the program instruction by instruction. The main method in this class handles the "translation" (using `TacScanner2`) of the source file into the corresponding `TacProgram2` object, prior to using the class's `execute` method to run the program.
9. Data structures:
   - The program itself will be represented by `TacProgram2`, which is essentially a list (ADT List) of `TacInstruction2` objects.
   - You will need to devise some mechanism to keep track of the values of the various variables as the program executes.
   - You will also need to think about how to maintain a list of the various program labels and the instruction to which each corresponds.
10. Input and Output: Use `java.util.Scanner` wrapped around `System.in` to handle input and `System.out` to handle program output.
11. It is assumed that program execution begins with the first instruction and proceeds until a `halt` instruction is encountered. Ordinarily execution proceeds from each instruction to the next, but may be altered by jump (conditional or unconditional). Note that each `TacInstruction2` object encodes the instruction type (assignment, read, goto etc.).
12. The completed application should be able to execute any TAC program specified on the command line.

    ```
    java TacExecutionEngine fact.tac
    ```

    You may assume that the TAC program supplied is syntactically valid.

Sample TAC Program

```
read x ;                    # read x;
t0 = 0;                     # if 0 < x then
t1 = x;
t2 = t0 < t1;
if (t2 == 0) goto l0;
t3 = 1;                     #   fact := 1;
fact = t3;
l1:                         #   repeat
t4 = fact;                  #      fact := fact * x;
t5 = x;
t4 = t4 * t5;
fact = t4;
t6 = x;                     #      x := x - 1
t7 = 1;
t6 = t6 - t7;
x = t6;
t8 = x;                     #   until x = 0;
t9 = 0;
t10 = t8 = t9;
if (t10 == 0) goto l1;
t11 = fact;
write t11;                  #   write fact
l0:                         # end
halt;
```