CS 240 Homework 6

Alex Vondrak

DUE: February 24, 2012

When high-level code (e.g., Java) is translated into low-level machine instructions, we often lose more details than we realize. The specifics depend on particular languages, machine architectures, operating systems, etc. But a fairly common feature of high-level programming languages—recursion—generally has to be implemented on the machine level using a *call stack*. The call stack keeps track of every method call (not just recursive ones), but for our purposes, we'll be looking at how we can use a stack to keep track of recursive method invocations.

In this homework, you'll simulate a call stack for two separate recursive definitions of the factorial function ($n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \cdots \times n$). One of these definitions will be the "regular" recursive definition that takes one parameter, and the other will be a tail-recursive version that takes two. The process of writing Java code to simulate low-level machine code isn't exactly straightforward, so for this homework, much of the code will be provided for you: `http://www.csupomona.edu/~ajvondrak/cs/240/12/winter/hw/hw6.java`. Notice a few things about this code:

- The file begins with the `LinkedStack<E>` implementation, much like the one you were asked to write for Homework 5 (hence this homework being posted a little late).
- The `CallFrame` class represents the data structure of the elements on the call stack. Every time a method is invoked, a *call frame* is allocated and pushed to the stack. When a method returns, the top frame (corresponding to the most recent method call) is popped. A call frame contains data specific to the particular method invocation, including the values of local variables and the point we're at in executing the method (i.e., the location of the current statement being executed).
- The `FactorialSimulation` class is where the various factorial function implementations live. There are two as-of-yet unimplemented methods, and it is your job to implement them. This is discussed below.
- Pretty much all the I/O has been taken care of for you in the `Homework6` class.

So, what are you supposed to do with this code? Well, `FactorialSimulation` has two sets of methods: `public int factorial(int n)` and `public int run()` correspond to the straightforward non-tail-recursive definition of the factorial function, while `public int factorialTR(int n, int r)` and `public int runTR()` correspond to the tail-recursive version. Because of the similar forms, this gives you a chance to both translate Java code into a simulation of the call stack, and to translate a simulation of the call stack into its original Java code.

To this end, you must implement two methods:

- `public int factorialTR(int n, int r)`, which must be a tail-recursive factorial implementation based on the simulation in `public int runTR()`. Include comments like the ones in the given `public int factorial(int n)` definition that indicate what statements the magic numbers[1] in `runTR` correspond to.
- `public int run()`, which must be a call stack simulation of the recursive definition provided by `public int factorial(int n)`. It will be very helpful to read the `runTR` code, understand how it's working, and mimic the general structure.

Input Format

Input consists of a series of Java `int`s, each representing a number $n \geq 1$ whose factorial will be calculated.

Output Format

Output is generally dictated by the provided code (as long as you call `this.report()` in all the right places). See the Sample Output and the provided code.

Input Sample

Test against the ACID input available at `http://www.csupomona.edu/~ajvondrak/cs/240/12/winter/hw/hw6.in`

Output Sample

Test against the ACID output available at `http://www.csupomona.edu/~ajvondrak/cs/240/12/winter/hw/hw6.out`

[1] Don't worry too much about the magic numbers in this assignment.