CMSI 585

PROGRAMMING LANGUAGES (GRADUATE LEVEL)

Fall 2005

Assignment 1101

This assignment represents a number of control flow programming experiments and exercises on selected languages, as well as a few conceptual questions.

Not for Submission

1. Read Scott Chapter 6.
2. You will need Andrew Wall's CUnit for this assignment; refer to the course Web site for the link, and look at my *gcd* and *roman* sample code for pointers on how to use it.
3. You should probably start writing your paper at this point. Feel free to show me drafts for feedback.

For Submission

As usual, submit code on hardcopy and by e-mail, and submit all other exercise responses on hardcopy. There are 3 programs to write this time, but they are all relatively short, and only one of them requires installation/setup of a new unit test framework (if you haven't done so already). The ML unit tests are homebrewed and provided in the *assertFactorial.sml* handout.

1. Write a module + unit test suite that explores expression evaluation order in the C compiler that you're using. Since we know that evaluation order is by default undefined, we won't do cross-unit-test-running technique on this one. We are using the unit test format in this case as an unambiguous picture of how your compiler is behaving. Here are the specifics:
   a. In *yourLastName_EvalOrder.h*, declare two functions, *int f()* and *int g()*.
   b. Implement these functions in *yourLastName_EvalOrder.c*. They can do anything you wish; the criterion is that they will expose evaluation order in your C implementation.
   c. Write a CUnit unit tester that uses *f()* and *g()* to determine the evaluation order policy used by your C compiler. Generally, this involves first calculating an expression in a way that forces evaluation order, then determining the result with an "all-in-one" version.
2. Based on the program you have written, answer the three questions in Scott exercise 6.4.
3. Fill out the *doLoops()* method that is stubbed out in the *loop.Loop* handout. The method should run three nested loops and maintain three *int* counter variables, one for each loop level. Each loop iteration starts by adding a new *Loop* object to the *List<Loop>* that will be returned by the function; that *Loop* object should have the loop level (e.g., 1 for the outermost loop) and the current values of each counter variable at that point. Then, the counter variable corresponding to that loop (e.g., *counter1* for the outermost loop) should be incremented. In the innermost loop, the following mid-tests are done after the counter for that loop level has been incremented:
   a. If the innermost counter is evenly divisible by 5, then leave the innermost loop.
   b. If the innermost counter is evenly divisible by 11, then leave the middle loop.
   c. If the innermost counter is evenly divisible by 14, then leave the outermost loop.

Write a JUnit unit test called *loop.test.yourLastName_LoopTest*. It should have a single test method that calls *doLoops()* and checks the *List<Loop>* that it returns for correctness.

For this one, we will use the same unit test/open source technique to see how we do in writing this code. As always, don't hesitate to e-mail me with any questions or clarifications.

4. Do Scott exercise 6.24 in ML, in two files: (1) define the *factorial* function in a file called *yourLastName_factorial.sml* (just the filename, not the function name — that should stay as *factorial*), (2) test your *factorial* function in *yourLastName_testFactorial.sml*, using the ML unit test functions provided in the *assertFactorial.sml* handout.
5. How would one determine whether or not a programming language implementation detects and transforms tail recursions?
6. Answer Scott exercise 6.27.