CSE305, Fall 2007 **Assignment 6** **Due Wed. Oct. 10**

This is a short problem set, before the **First Prelim Exam**, with will be held in-class on **Friday, Oct. 12**. The exam will cover Sebesta chapters 1–5, plus concepts of precedence and associativity in evaluating expressions from Chapter 7 which went with Chapter 3, and concepts of activation records, the run-time stack, and scoping from Chapter 10 which went with Chapter 5. Operationally the domain of the exam will be Assignments 1–5, plus perhaps some true/false or multiple-choice or brief-prose-answer type questions.

That is to say, the "Stack Language" and coding in ML on this Assignment 6 are *not included* per-se on the exam. But: working out the stack language problems may iron-in your knowledge of concepts from Chapter 3 and of pointers/references/bindings/lvalues/rvalues, while coding in ML might help you better understand the *grammar* of ML, which might be referenced on an exam question as it was on previous assignments.

Reading: For Friday and next week, read Chapter 6. With the way Java and C# since 2005 have overhauled OOP type systems even down to basic levels, there is a wealth of new material that relates to this chapter (and isn't *in* it since this edition is mid-2005). Hence I *may go quite slowly* through this material—I have to judge what to say now and what to leave for Chapter 12 and current OOP language-design issues at the end.

The Mon.+Tue. 10am recitations next week are still in Bell Hall, room **224** (not 242), as work on the Commons Conference Room is still stalled.

(1) Compile the following code (all lines except the declarations and print statement) into the "Stack Language" described on a class handout (copy also on the language-resources webpage), which has equivalent semantics to the Java Virtual Machine but operates differently. You may use the names `p`, `q`, `x`, and `y` in your stack code, but should be aware that the execution model would actually use the integers corresponding to them as per last week's homework. Also trace the stack at each step of execution, and show the changes in the storage objects again like you did last week. (30 pts. total)

```
#include "stdio.h"
void main() {
        int x, y, *p, **q;
        x = 5;
        y = x+3;
        p = &x;
        q = &p;
        (**q) = y + x;
        (*q) = &y;
        (**q) = y - x;
        printf("Final value of y is %d\n",y);
}
```

(2) Write in ML a function `solve(a,b,c)` that returns a pair `(n,r)`, where `n` equals the number of distinct roots of the equation $ax^2 + bx + c = 0$, and r is the larger root if there are two, the unique one if the roots are equal, or 0 if none. The square-root function can be accessed as `Math.sqrt(x)`, short for `Real.Math.sqrt` since `open Real;` is automatic at ML startup, and you can just multiply rather than use `Math.pow(x,y)` for powering. Note that ML enforces a rigid distinction between the `int` and `real` types, and that `real` does not provide an equality test since finite-precision makes that operationally dicey. (You may hand-write your code in hardcopy—this is *not* for online submission. 18 pts., for 48 total on the set.)

The following snippet of ML grammar has all you need, and actually you can ignore everything from `case` onward.

```
DEC ::= val PAT = EXP {and PAT = EXP}[;]
  | fun FBIND {and FBIND}[;]
FBIND  ::=  ID APAT {APAT} = EXP {"|" ID APAT {APAT} = EXP}
APAT ::=  CONST                        any constant of any type
  | ID | _                             ID is *written to*, not read!
  | "(" PAT{, PAT} ")"                tuple pattern
  | "[]" | "[" PAT{,PAT} "]"          list pattern of fixed size
EXP ::= CONST | ID | PREFIXFN | PREFCON
  | #LABEL                            #2 (a,b) = b, #c {c=3, e=4} = 3
  | "("EXP{, EXP}")"                  tuple.  Note "(EXP)" is legal
  | "("EXP{; EXP}")"                  sequence-of-"statements"
  | "["EXP{, EXP}"]"                  list of known length
  | let {DEC[;]} in EXP{;EXP} end  Makes a scope. Last EXP has no ;
  | EXP EXP           Includes PREFIXFN EXP.  Just space, no comma!
  | EXP INFIXFN EXP   See top-level infix ops below, precedence 4-7
  | if EXP then EXP else EXP          Just like "EXP ? EXP : EXP" in C
  | case EXP of MATCH                 Often needs (...) around it

MATCH::=  PAT => EXP {"|" PAT => EXP}
PAT  ::=  APAT                         atomic pattern
  | ID PAT                             pattern with a constructor
  | PAT :: PAT | PAT ID PAT            includes list pattern x::rest
PREFIXFN::=  op INFIXFN  |  !  |  ~  |  not  |  abs  |  ID
INFIXFN ::=  *  |  /  |  div  |  mod |  +  |  -  |  ^  | :: | @ | etc.
```