PL 92b qual

1. Statement and program equivalence
    a. S1 = x := 0; while(x);
       S2 = x := 0;
       These are L1 equivalent, as both terminate (given the L1 and L2 from part b). However, These are not L2 equivalent as shown by:
       cobegin
           x := 0;
           and
           x := 1;
       coend
       And the L1 equivalent replacement
       cobegin
           x := 0;
           while(x);
           and
           x := 1;
       coend
       In words, we have now found L1 equivalent statements that are not L2 equivalent. This happens because under L2, the execution order is not fixed, so non-determinism can cause the code to not terminate.
       Assuming S1 and S2 contain only elements that are members of L1, then if they are L2 they are definitely L1, as L2 is a superset of L1. However, should S1 and S2 contain elements that are not members of L1, then they are not members of L1.

    b. Use the definitions for S1 and S2 given in part a. L1 equivalence is kind of obvious. A simple assignment will terminate, and a while(0) will also terminate, so they are L1 equivalent. The reason they are not L2 equivalent was shown above in part a.
2. Prolog vs. ML
    Prolog unification – tries to unify each variable with each in function. Exits once it finds a match.
    ML pattern match – matches types, assigns variables, checks if valid
    ML is easier – by restricting variables to appear only once, you don’t have to do the unification step.
    If we allow variable names to appear more than once, then ML would need to do unification and store multiple copies. This would give us a logic programming language.
    If we extended ML in this way, we would get an easy translation from Prolog to ML. Basically just need a syntax change, but no variable changes or logic changes.
3. Compiling into C
    a. Advantages: high level language code generation is easier than assembly; good C compilers already exist; portable; binary compatability with existing C code.
       Disadvantages: mapping between languages could be tricky; extra level of compilation; debugging requires back compilation to original language from C; high order functions, garbage collection, and exceptions; inefficient or bulky code could result

    b. Advantages: common code base; use most natural language for task; cross language calls

    c. Difficulties: garbage collection; high order functions; true polymorphism; untyped vs typed interface.

    d. Function problems: high order functions need closures

    e. Compiler tradeoffs: need closures; your compiler must handle the high order functions – more work for you. C stack can do locals.