PL 1993 qual

1. Subtyping and Polymorphism
    a. Subtyping – substitutivity. If A is a subtype of B, then A can replace B without type error.
        i. Advantages – uniform operations over multiple types of data, can add functionality with minimal modification to system.
        ii. Implementation cost – compilation complexity if subtyping relies on inheritance
    b. Polymorphism – dynamic method lookup. Function that can be applied to arguments of more than one type.
        i. Advantages – code reuse (Templates at the user and programmer level, polymorphic functions at the code level). Dynamic lookup – all objects call same method 'name'
        ii. Implementation cost – level of indirection is slower (compiler, runtime). In C++, templates generate multiple copies of the code.

2. Smalltalk vs. C++
    a. Smalltalk:
        i. Properties – dynamic typing, implicit subtyping (not linked to inheritance), everything is an object (int, bool), programmer template and method lookup, no multiple inheritance.
        ii. When useful – flexibility, easier to add things, stability, more correct programs
    b. C++:
        i. Properties – static typing, explicit subtyping through inheritance, not everything is an object, vtable, multiple inheritance.
        ii. When useful – speed, don't pay for unused features (object lookup), existing code

3. Prolog

4. Closures vs. objects
    a. Closure – function and its execution environment
    b. Object – data (hidden) plus functions operating on the data
    c. Difference – no inheritance for closures, object is its own environment
    d. Closures can simulate objects (except subtyping and inheritance) by creating an object when you need to return the closure.
    e. Can objects simulate high order functions? We speculate you can simulate a closure with an object, and then simulate high order functions with the closure. In fact, see http://www.cs.hmc.edu/~keller/Polya/ for an implementation of high order functions in C++.

5. Lisp and Pascal combining
    a. Problems – type checking, memory allocation, variable access, moving data between procedures, memory references, interpreted vs. compiled
    b. Fixes – define an RPC like interface between Lisp parts and Pascal parts. This requires everything to be converted into an 'intermediary' form to cross boundaries between languages. You could also try compiling Lisp or interpreting Pascal.

6. Eliminating the run-time stack in a compiler.
    a. Reasonable when – efficiency doesn't matter, high order functions already exist, Lisp, ML
    b. Losing proposition when – need efficiency, need explicit memory control, don't need to return functions, Pascal, C++