

Comparison of CS 172 to “Computing Curricula 1991”

R. Brown, July 30, 2003

Recurring Concepts

(Note: Numbers below refer to lecture number in the accompanying example syllabus. Students use “Scheme” language for programming in the course.)

Binding 1 (naming of files, printers, computer accounts, computers, programs, etc.); 2 (**define**); 6 (associating a function definition with a spec); 7, 16–17 (local names); 8 (role of naming in recursion); 13 (associating a named argument with an accumulating answer); 16 (scope of name bindings); 16, 18 (naming of “helper functions”); 21 (file names); 28 (implementing classes from specs).

Complexity of large problems 1 (Scheme vs. **emacs** editor vs. UNIX as description of local Scheme system); 11 (linear vs. nonlinear recursion); 14–15 (demonstration and trace-based analysis of non-linear recursion vs. linear iteration); 27–30 (object-oriented programming for managing large-scale projects).

Conceptual and formal models 2 (architecture diagram); 3 ff (specification of functions); 3 ff (formal standard for Scheme language); 8–9 (understandings of recursion through problem-solving vs. syntax vs. computational trace; stopping conditions for recursion); 14 (stack-frame model of implementation of recursion vs. iteration); 17, 23 (environment model of bindings); 22–23, 25 (state model of memory, including box-and-arrow diagrams for lists); 24, 27 (programming paradigms); 27–28 (object-oriented model of programming).

Consistency and completeness 3 (completeness vs. correctness of specs); 3, 5, 15, etc (using verbal specs, invariant assertions, iteration/loop invariants to verify program correctness).

Efficiency 11 (traces of linear vs. nonlinear recursion computations, counting numbers of comparisons/function calls, etc.); 14 (iteration vs. recursion, counting stack frames).

Evolution 2, 6, 18, 28 (expanding understanding of capabilities of functions, from predefined actions to user-definable reusable operations to higher-order customizable abstractions to objects with state); 2, 16, 28 (widening notion of binding of names, from global bindings to local bindings to closures for encapsulation of object state variables); 2, 17, 22–23, 25, 28 (progressive development of conceptual memory model); 3 ff (increasing completeness of specs as technical issues emerge); 3, 4, 7, 19, 21, 28 (alteration of concept of spec to accommodate procedures, special forms, functions that return functions, functions with state changes, specs for classes in object-oriented programming); 20, 29 (code maintenance as rationale for procedural abstraction, object-oriented programming).

Levels of abstraction 1, 2, 14, 17, 21, 22–23, 25 (levels of hardware description); 4, 25 (understandings of lists at high level vs. low level); 6, 28 (specification vs. technical details of code that implements a spec); 8–9 (understandings of recursion conceptually vs. low-level computational trace); 13, 15 (reasoning out correct iterative algorithms from spec and iteration invariant, without tracing); 16, 20, 28 (hiding implementation details through local bindings of helper functions, applications of procedural abstractions and encapsulation and

message passing in object-oriented programming); 18–19 (capturing recurring computations in a procedural abstraction); 29 (levels within a class hierarchy with inheritance).

Ordering in space 1 ff (use of network computing, with machines in CS lab, files in library, perhaps remote login from dorms); 4, 25 (consecutive elements in a list, as conceptual elements and as `cons` pairs in memory, consecutive memory locations in a `cons` pair); 9 (tracing consecutive recursive function calls); 14 (accumulation of stack frames); 16, 18, 28 (grouping of related code in a system into a package, e.g., helper functions for tail recursion and state variables/methods for an object); 16, 27, 28 (locality of binding of names; scope of a binding); 21 (input/output to disk files).

Ordering in time 1 ff, 21 (time-ordering of sequential evaluation, e.g., binding with `define` before using the bound name, `begin`); 3 (order of evaluation of arguments in procedures vs. special forms); 6, 18, 28, 29 (immediate vs. delayed evaluation in `lambda`, special forms `let` and `letrec`, higher-order functions, constructors for objects, delegated methods); 9, 14 (order of execution of recursive and iterative function calls).

Reuse 3, 6, 21, 28 (adhering to the Scheme standard and implementing specs for portability); 6, 18, 21 (capturing algorithms and state changes using functions and procedural abstractions); 22 (assignment viewed as reuse of memory locations); 26 (garbage collection for “recycling” of unused memory); 27–30 (object-oriented programming for reuse of algorithms, state-variable configuration, inheritance of classes, etc.); 28 (multiple use of a file of definitions, e.g., `account.scm` file for defining *Account* class, `send` message-passing primitive, etc.).

Security 1 (UNIX passwords for access to lab machines); 12 (error checking in function definitions); 27 (encapsulation to prevent unstructured modification of state variables).

Tradeoffs and consequences 3, 18 (reliability and maintainability of functions and procedural abstractions); 12 (consequences of incorrect code, inappropriate applications of functions, etc.); 14 (recursion vs. iteration); 16 (local vs. global bindings of helper functions); 20 (desirable qualities for procedural abstractions; tradeoffs in deciding whether to use procedural abstraction); 27 (rationale for object-oriented programming).