

CMSI 585

PROGRAMMING LANGUAGES (GRADUATE LEVEL)

Fall 2006

Final Review Sheet

The final will take place on Tuesday, December 12, 2006, at 6:30pm, *in the Keck lab*. It will be open book, notes, handouts, *and* computer; you may use either your own laptop or one of the lab's workstations.

Covered Material

The final covers the following areas, including all handouts and sample code that have been distributed in support of this content:

- Scott Chapters 1, 2, 3, 6, 7, 8, 9, and an overview of 12
- Working knowledge of C, C++, Java, JavaScript, ML, and Perl
- Some familiarity with unit testing frameworks in these languages

Specific Knowledge Areas

All of these topics are fair game for the final:

- Building and execution models of different programming languages (Compiled? Interpreted? Type of target code? Run-time environment?)
- Typical (or possible) phases in the process of moving from source code to program execution
- Definition and interpretation of regular expressions and context-free grammars
- Familiarity the scanning and parsing processes, as well as a working knowledge of how automated scanner and parser generators work
- Concepts and definitions related to naming, scopes, and binding, and the ability to reason about different scenarios based on these concepts (e.g., What languages tend to have which scoping rules? When does binding or unbinding occur? To what object will a symbol resolve, if at all? What module mechanisms exist for different programming languages?)
- Properties of modules (open, closed, etc.) and their roles when programming “in the large”
- The two paradigms for modules — module-as-manager, module-as-type — and how code in each of these paradigms is structured
- Concepts, definitions, and variations on expression evaluation, assignments, and variables

- Evolution of control flow constructs; the constructs themselves: selection, iteration, recursion, nondeterminacy; implementation issues with these constructs; tail recursion
- Data types: type checking rules (equivalence, conversion, coercion, inference); the “type zoo,” and any language design or implementation issues that accompany these types (primitives, records, arrays, strings, sets, pointers, recursive types, lists, streams)
- Subroutines: calling sequences, parameters, generics, closures, exceptions
- Object orientation: definitions, evolution from abstract data types and modules, dynamic method binding, implications at runtime
- Concurrency: general issues, language constructs to accommodate concurrent programming

Sample Tasks and Questions

The following represent the types of questions or tasks that you may be asked to accomplish:

- Diagram the build or execution model of some programming language
- Given some code, identify the type of error
- Given a grammar, identify strings that are in (or not in) that grammar's language
- Given a specification, write the grammar
- Identify an ambiguous grammar; decide between alternative grammars for the same construct
- Given a code fragment, identify events relating to object and binding creation
- Given a code fragment, identify possible unexpected results or degenerate cases (e.g., due to side effects, speed or memory issues, etc.)
- Given some language feature or construct, discuss language design or implementation issues that may arise from including it in the language
- Rewrite a recursive algorithm as tail-recursive

- Interpret execution possibilities on a nondeterministic code fragment
- Data type-related tasks for a code fragment (e.g., what is equivalent, what the final type of an expression is, where does coercion take place, etc.)
- Discuss implementation or other language design implications for certain data types, such as enumerated, subrange, or derived types
- Delineate the sequence of events for a specific subroutine call
- Describe what may or may not happen to parameters in a particular subroutine call
- Discuss language design or implementation issues with passing subroutines as closures
- Discuss language design or implementation issues with generic subroutines and modules
- Given an object-oriented code fragment, determine or infer specific sequences of events or binding resolutions, such as constructor calling order or method binding
- Given a code fragment with multiple inheritance, provide the possible interpretations or specific design choices that can be made on what gets inherited (or not)
- Compare advantages, disadvantages, or implementation issues regarding different approaches to concurrent programming language design
- Interpolate possible execution sequences for coroutines or other concurrent code fragment