

CS 264 Programming Language Implementation

Prof. R. Fateman : Spring, 2004

About the course

This course is one of three graduate courses that follow the undergraduate CS164 (Programming Languages and Compilers). CS263 concentrates on language design, semantics, and related issues. CS264 is on advanced language implementation. The third course, CS265, covers more topics in implementation and in particular concentrates on compiler optimization and code generation. There are additional topics in parallel languages covered in CS267, although there the emphasis is using these languages in applications. There is also some coverage of implementation of languages for artificial intelligence in CS 283.

Prerequisites

Students are expected to have taken CS164 or an equivalent undergraduate course in compiler implementation. You are presumed to be able to invent a grammar and write a parser for a simple programming language. You are expected to have some exposure to at least two different programming languages, and substantial programming experience in at least one.

Computer use

Graduate students in EECS or CS should have accounts on UNIX or other machines suitable for reading web pages, writing programs and reports, executing programs, etc. Undergrad EECS or CS majors should also have suitable accounts. If you don't fit in these categories (e.g. a Math major?) and need an account, see me.

Meeting times etc.

CS264 is 4 units and traditionally has had a stronger project/programming component than the other CS26X courses.

Class meeting times: 9:30-11:00 in 310 Soda. Other times may be scheduled with sub-groups on projects. There is no teaching assistant.

Office hours: 2:00-4:00 Tu, Thurs or by appointment, room 789 Soda.

Class home page: <http://www.cs.berkeley.edu/~fateman/264/> will have useful material for the class.

Books

None required: we will post material on the web page and distribute printed copies sometimes. The following books will occasionally be mentioned, and access to them may be helpful.

- A. Appel, *Modern Compiler Implementation in {Java, ML, C}*, Cambridge University Press, various dates.
- A. Aho, R. Sethi, J. Ullman *Compilers: Principles, Techniques and Tools*, Addison-Wesley (Red Dragon)

- K. Cooper, L. Torczon, *Engineering a Compiler*, Morgan Kaufmann, 2004.
- Daniel P. Friedman, Mitchell Wand, Christopher T. Haynes, *Essentials of Programming Languages*, MIT Press 1992.
- S. Kamin, *Programming Languages: An Interpreter-Based Approach*, Addison-Wesley, 1990. This is out of print, but a total re-write effort lead by Norman Ramsey is in progress (2003). If we ever space in our curriculum for an undergraduate version of a programming language (not “compiler”) course, this might be a suitable text.
- Steven Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.
- Peter Norvig, *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann.
- Michael L. Scott, *Programming Language Pragmatics*, Morgan Kaufmann, 2000. .
- Glynn Winskel, *The Formal Semantics of Programming Languages, An Introduction*, MIT Press, 1993.

Topics

The CS264 catalog description includes these topics: Compiler construction, lexical analysis, syntax analysis, semantic analysis, code generation and optimization, Storage management, Run-time organization.

In reality, we expect to spend some time rapidly reviewing CS164 material, as well as some material from CS61abc. We do not expect to dwell on this material, however.

We view our subject as Advanced Language Implementation referring both to advanced languages (Lisp, Scheme, ML, Prolog, Smalltalk, Haskell) as well as (advanced) implementation.

While there is considerable overlap with conventional language issues (register allocation, data-flow analysis) here there is more concern with run-time (dynamic) type management, object/storage management, parallelization. For reference, there is a collection by Peter Lee on this topic, but it is now about 10 years old. We should add to Lee’s selection other non-traditional topics like non-determinism, order/kinds of evaluation, source-language transformations (macros, recursion removal, Continuation Passing Style), interpreters, computing with types, variants on object orientation. The Friedman-Wand-Haynes text (EOPL) has some interesting insights, and two other books (Kamin, Norvig) both contain rather pragmatic treatments of interpreters. There are also rather new areas for study including the complications implied by software and hardware pipelining, VLIW architectures (The Intel IA-64), and explicitly parallel languages (e.g. at Berkeley, www.cs.berkeley.edu/projects/titanium), perhaps NESL (www.cs.cmu.edu/~blelloch), Fortran-90, as a start. The interaction of compilers and floating-point computation is another topic of interest. Furthermore, we hope to be able to cover some topics, and have student reports on, in no particular order:

- How do language standards happen?
- Interrupts, exceptions, out-of-order processing
- Languages and features for parallel or distributed processing; (current state of the art?)
- Make/defsystem/asdf/CVS/metrowerks: the externalities of building a large system from source;
- implementation of logic programming,
- libraries and access to peculiar semantics: e.g. arbitrary precision arithmetic and scientific programming;
- supporting legacy languages (COBOL, anyone?)

- The interactive environment model (Erik Sandewall, Comp. Surveys vol 10 no. 1.)
- Scripting languages: Perl, Tcl, Python, ?
- Profiling tools: Intrusive, non-intrusive random sampling, hardware sampling: PAPI, SBC Lisp?
- Handwritten, spoken, languages? \dot{z}
- Is there such a thing as Language-motivated Architectural feature (as opposed to “good idea”?)
- Smart Memory and PL. (IRAM) .. ? Applications like FFT, hash, sort; languages like Fortran C, (Yelick?) MPI, Split-C, Titanium, UPC

Grades

Your grade in this course will depend on homework assignments (problems, programs), class participation, presentations and (mostly) a project. There may be a take-home final. In conformance with department policy, we ask that all students who regularly attend (even as “auditors”) officially register. Auditors should register for a P/NP or S/U grading option, with the understanding that regular attendance and participation in class will suffice for receiving a P or S. Completion of assignments or exams will not be required. Registering all attendees is the only way the department (and the instructor) gets teaching credit, and appears to be in line with the policy of other departments.

In the past, projects have generally been significant programs comprising extending techniques from the readings to new applications, experiments on improved algorithms, data structures, or significant benchmarking and analysis.

Sampler of Possible Projects

1. Repeat selected parts of a classic study by Knuth: “An Empirical Study of Fortran Programs,” (Knuth) Software— Practice and Experience 1 (2) 1971! (group project, probably)
Can we do this on (say) Java or C? or parallel programs? (cf. Z. Shen’s “Empirical study of Fortran for Parallelizing Compilers” IEE Trans on Parallel Comp system 1 (3) (1990).
We have huge numbers of “good” programs in Netlib; we can look at the source code for UNIX, Netscape, etc.
2. CPS and the transformation of functional programming style to imperative style: Can we develop a critique of functional programming that holds water from an efficiency perspective? (EOPL, Friedman, Wand, Haynes).
3. GC vs. Stack vs other kinds of memory management: (Appel paper on GC, other views, ancient history on memory allocation, new results on conservative GC for C and Java.)
4. Scripting languages and mobile computing. Tcl, scheme/Lisp, Java, Javascript, Active-X, Obliq (Cardelli, DEC). What are the important issues and how are they being addressed? (application areas: Web scripting, emacs, CAD systems, middleware) An empirical study? See also documents as programs (below).
5. Languages for secure computing. Why is (isn’t) Java secure? Could you make a secure and yet still useful version of another language?
6. Languages for reliable programming. Studies of error rates for C programmers are available. How could language design affect fault frequency? (Compare to, for example, Ada).

7. Parallel languages. Pick one or two important issues: see how are they being addressed, come up with a variation on a solution, perhaps an empirical study?
8. Documents as programs. As programmers we tend to think of material as programs + documents. There is a trend how to look at documents as a substrate for programs. This can be hazardous (e.g. Microsoft Word macro-viruses), or quite useful: html plugins, javascript, etc. Other examples would be “Scientific Workplace” or our own multivalent document (MVD) design for www.elib.cs.berkeley.edu. Emacs blurs the distinction as well. Make some sense of what has been done, and propose a clean extensible alternative. The role of XML and (in mathematical applications) MathML has come to the fore as browsers have adopted it, but most applications do not use XML.
9. Parsers for only approximately correct input seem required for use by browsers, since HTML is often incorrect. Develop an approximate parser for XML. Develop a parser for spoken programming (actually, a parser for mathematical expressions would be neat).
10. Look at tools for creating parsers, manipulating grammars, producing scanners: Given that computers are extravagantly faster and have more memory, make a case or improving and/or ignoring these tools.