Course Information

Course structure for Fall 2002

This semester the course will focus on compilation of functional programming languages. Important topics are intermediate representations, optimizations at several levels of intermediate representation, code and data layout, memory management, generation of executable code.

In addition to lectures and occasional written homework, students will work on a semester-long compiler project, divided into several stages. The goal of the project is to develop a fully functional translator from a functional programming language to an executable form.

The compiler project will require a significant implementation effort. It is intended that each stage should be about a week's effort.

The implementation language will be chosen by each participant individually. All project stages will be defined as input/output specifications without prejudice to the implementation language. An important aspect of the project will be to report on the convenience or inconvenience of using a particular implementation language. However, the specifications themselves will be expressed in ML, and class discussion of compiler implementation will use ML as the implementation language.

Prerequisites in detail

No specific courses are prerequisites for this class.

Students should be familiar with several high-level programming languages, preferably including representatives of the procedural, object-oriented, and functional programming families, so that they can appreciate the purpose and the tasks of a compiler in general.

Students should be experienced programmers able to develop very large programming projects implemented in some programming language. The ability to keep up with deadlines is important.

There are some specific topics that will be assumed to form the background of each participant:

- functional programming in general
- understanding recursive data types, recursive functions to compute over them, and structural induction to prove things about them
- a language in the Lisp family (dynamically-typed)
- a language in the ML family (statically-typed)
- an exposure to logic programming and unification
- computer organization and architecture
- operating systems
- machine language and assembly language programming, especially for a RISC architecture
- the C programming language
- an exposure to programming language implementation, including interpreters and compilers
- the practice of scanning and parsing

Lectures

Mondays & Wednesdays, 2:30 - 3:45, in Tapy 218.

Instructor

Darko Stefanovic, office FEC 345C, phone 2776561, email darko@cs.unm.edu — office hours Mondays 3:50-5:00, or by appointment.

Teaching assistant

None

Attendance

Your attendance at lectures is mandatory.

Grading

The grade will be determined as follows:

- Programming project 80% (developed and graded in stages)
- Exams 20% (10% midterm exam, 10% final) The format of the exams remains to be decided; take-home exams are a possibility, and so are essays.

You are expected to attend class regularly, read the assigned reading before class, and participate in class discussion. Class participation is not directly graded, but may decide borderline cases.

Programming assignments

The textual layout of programs must be logically sound and aesthetically pleasing. The names used must be descriptive. Code comments and additional documentation must accompany programs, and must provide informal argumentation that the program satisfies the specification. (Occasionally you will be asked to provide a formal correctness proof.) For additional detail, see Norman Ramsey's guidelines.

Programming assignment hand-in policy

Programming assignments are to be submitted on-line. Detailed instructions will be provided with each assignment. Hand-drawn diagrams, if needed, should be submitted in class. Late programming assignment submissions will be penalized $3n^3$%, where n is the number of days late.

Cooperation and cheating

You are encouraged to *discuss* the project with classmates and the instructor. However, *do not look at or copy another student's solution to a homework or project*. If a problem appears too difficult, or you lack the background to solve it, you are expected to talk to the instructor promptly. Once you have the background necessary

to solve a problem, you must provide your own solution. Exchanging homework or project solutions is cheating and will be reported to the University administration; students involved will not be permitted to continue in the class. You are responsible for exercising due diligence in protecting your homework files from unauthorized access. In case two or more students present essentially similar homework, all involved students will be reported to the University administration. Homework must be accompanied by the following statement: "*I pledge my honor that in the preparation of this assignment I have complied with the University of New Mexico Board of Regents' Policy Manual, including Section 4.8, Academic Dishonesty.*" The manual is available at `http://www.unm.edu/~brpm/index.html`.

Textbooks

Titles marked with an asterisk have been ordered and should be available at the UNM bookstore.

Required reading

There is one required textbook for this class:

* Andrew W. Appel: *Compiling with Continuations* Cambridge University Press, 1992, ISBN 0-521-41695-7.

Optional reading (background material: garbage collection)

* Richard Jones and Rafael Lins: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* John Wiley, 1996, ISBN 0-471-94148-4.

Optional reading (newer books on compiler construction in general)

Michael L. Scott: *Programming Language Pragmatics*, Morgan Kaufmann, 2000, ISBN 1-55860-442-1.

* Andrew W. Appel: *Modern compiler implementation in ML* Cambridge University Press, 1998, ISBN 0-521-58274-1.

Reinhard Wilhelm and Dieter Maurer: *Compiler Design*, Addison-Wesley, 1995, ISBN 0-201-42290-5.

Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs and Koen G. Langendoen: *Modern Compiler Design*, John Wiley, 2000, ISBN 0-471-97697-0.

David A. Watt and Deryck F. Brown: *Programming Language Processors in Java*, Prentice Hall, 2000, ISBN 0-13-025786-9.

Keith Cooper and Linda Torczon: *Engineering a Compiler*, Morgan Kaufmann, expected publication date January 2003.

Optional reading (older books on compiler construction in general)

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman: *Compilers : principles, techniques, and tools* Addison-Wesley, 1986 (Reprint with corrections March, 1988), ISBN 0-201-10088-6.

C. N. Fischer and R. J. LeBlanc: *Crafting a Compiler*, Benjamin/Cummings, 1988.

Optional reading (optimizing compilation)

Steven S. Muchnick: *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997, ISBN 1-55860-320-4.

Michael Wolfe: *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1996, ISBN 0-8053-2730-4.

Randy Allen and Ken Kennedy: *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*, Morgan Kaufmann, 2001, ISBN: 1-55860-286-0.

Optional reading (functional language implementation)

Simon Peyton Jones and David R Lester: *Implementing Functional Languages: a tutorial*, available online.

Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes: *Essentials of Programming Languages*, MIT Press, 1992, ISBN 0-262-06145-7. (cross-listed as McGraw-Hill ISBN 0-07-022443-9)

David A. Watt: *Programming Language Concepts and Paradigms*, Prentice Hall, 1990, ISBN 0-13-728874-3.

Michael J. C. Gordon: *Programming Language Theory and its Implementation*, Prentice Hall, 1988, ISBN 0-13-7304170-X.

A J T Davie: *An Introduction to Functional Programming Systems Using Haskell*, Cambridge University Press, 1992, ISBN 0-521-27724-8.

Anthony J. Field and Peter G. Harrison: *Functional Programming*, Addison-Wesley, 1989, ISBN 0-201-19249-7.

Optional reading (background material: lambda calculus)

H. P. Barendregt: *The Lambda Calculus: Its Syntax and Semantics, revised edition*, Elsevier North-Holland, 1984, ISBN 0-444-87508-5.

Joseph Stoy: *Denotational Semantics*, MIT Press, 1977, ISBN 0-262-69076-4.

Optional reading (background material: unification)

Krzysztof R. Apt: *From Logic Programming to Prolog*, Prentice Hall, 1997, ISBN 0-13-230368-X.

Optional reading (background material: functional programming)

Richard Bird: *Introduction to Functional Programming using Haskell*, Prentice Hall, 1998, ISBN 0-13-484436-0.

Paul Hudak: *The Haskell School of Expression*, Cambridge University Press, 2000, ISBN 0-521-64408-9.

Richard Bird and Oege de Moor: *Algebra of Programming*, Prentice Hall, 1997, ISBN 0-13-507245-X.

Optional reading (background material: modern computer architecture)

* John L. Hennessy and David A. Patterson: *Computer Architecture: A Quantitative Approach*, third edition, 2002, ISBN 1-55860-596-7.

Optional reading (background material: programming in ML, for those using ML as the implementation language)

Jeffrey D. Ullman: *Elements of ML Programming, ML97 Edition*, Prentice Hall, 2000, ISBN 0-13-790387-1.

Lawrence C. Paulson: *ML for the Working Programmer, 2nd edition*, Cambridge University Press, 1996, ISBN 0-521-56543-X.

Michael R. Hansen and Hans Rischel: *Programming Using SML*, Addison-Wesley, 1999, ISBN 0-201-39820-6.

Optional reading (background material: other)

R. Tennent: *Principles of Programming Languages*

Ryan Stansifer: *The Study of Programming Languages*, Prentice Hall, 1995, ISBN 0-13-726936-6.

To probe further

John C. Reynolds: *Theories of Programming Languages*, Cambridge University Press, 1998, ISBN 0-521-59414-6.

David A. Schmidt: *The Structure of Typed Programming Languages*, MIT Press, 1994, ISBN 0-262-19349-3.

John C. Mitchell: *Foundations for Programming Languages*, MIT Press, 1996, ISBN 0-262-13321-0.

Benjamin C. Pierce: *Types and Programming Languages*, MIT Press, 2002, ISBN 0-262-16209-1.

Lecture Plan Overview (subject to change)

We are planning for 14 weeks of 2 lectures each.

- Organizational meeting (in the summer)
- Lambda calculus
- Core lambda language
- Data representation
- Simply-typed lambda calculus
- Polymorphism
- Interpreter for core language
- Passage to restricted forms of core language (and simpler interpreters)
- CPS (and simpler interpreters)
- Optimizations
- Closure conversion (and simpler interpreters)
- Lifting
- Native code generation
- Classical optimizations
- The run-time system: garbage collection and OS interface