Preliminary version of 18 December 2009

Course Information

Course structure for Spring 2010

A compiler for any high-level language is a large piece of software consisting of many different phases that express different program analysis, transformation, and optimization algorithms. Underlying these is a significant body of theoretical work. In addition, the variety of modern programming languages require specialized approaches to implementation (for instance, Java requires a compiler to be present while the program is running in order to load classes dynamically). Therefore, building a complete compiler for an entire programming language is a major undertaking. A single class cannot cover all aspects of "Advanced Topics in Compiler Construction"; it is intended that this course will be given repeatedly, and each time core concepts of compilation will be reviewed, but the focus will shift to different compiler components.

This semester the emphasis will be on the "middle-end" stages in the compilation of strict functional programming languages. The topics are intermediate representations and code transformations. In particular, we will study representations of control flow, the continuation-passing style and the CPS transformation and its relatives, the A-normal form and the static single assignment form. We will examine how higher-order functions can be efficiently implemented. In addition to their use in compilation, the transformations we will study can be viewed as programming techniques, and we will look at examples of their use.

Another area of interest is the definition and implementation of programming languages via abstract machines and virtual machines (bytecode compilation). Again, continuations play a big role in this development.

Prerequisites in detail

No specific courses are prerequisites for this one.

Students should be familiar with several high-level programming languages, so that they can appreciate the purpose and the tasks of a compiler.

Students should be experienced programmers able to develop large programming projects implemented quickly. 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, and Scheme, ML, or Haskell in particular
- understanding recursive data types, recursive functions to compute over them, and structural induction to prove things about them

- familiarity with computer organization and architecture, operating systems, machine language and assembly language programming, and the C programming language

Lectures

Tuesdays and Thursdays 9:30-10:45 location TBA

Instructor

Darko Stefanovic, office FEC 345C, phone 2776561, email darko — office hours TBA

Teaching assistant

None

Grading

The grade will be determined as follows:

- Programming projects 50%
- Homework and oral presentations 20%
- Mid-term exam 25%
- Class participation 5%

You are expected to attend class regularly, read the assigned reading before class, give occasional oral presentations, and participate in class discussion.

Programming assignment hand-in policy

Programming assignments are to be submitted on-line. Detailed instructions will be provided with each assignment. Late programming assignment submissions will be penalized $2n^2$%, where n is the number of days late.

Textbooks

The principal text for this class is *Semantics Engineering with PLT Redex* by Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt, MIT Press 2009. Additional reading material (tutorials and research papers) will be provided as needed.

Topics

Boldface: topics emphasized this term.

- Introduction to compilation
- The structure of compilers
- Front-end design
- Back-end design
- Common issues in the compilation of functional languages
- Compilation of strict functional languages
- Intermediate representations for strict functional languages
- Front-end design for strict functional languages
- Modules, functors, and their implementation
- **Abstract machines for lambda calculi**
- **Representing and analyzing control flow: CPS, ANF, SSA, and CFA**
- Parametric polymorphism and its implementation
- **Higher-order functions, closure conversion, and defunctionalization**
- Back-end design for strict functional languages

Lecture plan (tentative)

1. Organizational; Overview of core topics via examples that go all the way from source to machine code
2. Continue overview of core topics
3. Common compiler structure; Functional language compilation; Strict functional language compilation
4. Register allocation (a back-end topic)
5. Front-end design: Scanning; Recursive descent parsing; LL(1) parsing
6. Combinator parsing

7. Monadic parsing combinators; Parsec
8. Review of lambda-calculus
9. Call-by-value lambda-calculus and its definition by structural operational semantics
10. Abstract machines using the notion of evaluation context
11. Abstract machines using continuations; CEK
12. Tail calls; CPS transformation
13. Implementation(s) of the CPS transformation
14. Closure conversion
15. Review of typed systems; Simply-typed lambda-calculus; PCF; More extensions (records, variants, let)
16. Definitional interpreters
17. Review of denotational semantics; Evaluators for lambda-calculus; Continuation-based semantics for jumps
18. Defunctionalization
19. CPS in a typed setting
20. ANF
21. ANF implementation
22. Code generation and optimization for strict functional languages
23. Implementation of polymorphism; Representations for data
24. Representations for closures
25. Monadic code transformation
26. Control-flow analysis
27. Use of control-flow analysis in code optimization, closure conversion, and type reconstruction
28. Krivine's abstract machine; Landin's SECD as abstract machine and as virtual machine
29. Correspondences between evaluators and abstract machines
30. Certifying compilation; Certified compilers