Principles of Compiler Construction

Dr Mayer Goldberg
September 5, 2016

Contents

1 Course Objectives 1
2 Course Requirements 2
3 Detailed Syllabus 3
4 References 6

• Course number: 201-1-2061
• Mandatory for undergraduate CS and SE students
• Credits: 4.5
• Course site: http://www.little-lisper.org/website/comp.html
• Prerequisites: Principle of Programming Languages (202-1-2051), Automata & Formal Languages (202-1-2011), Architecture (202-1-3041)

1 Course Objectives

• Gain additional insight into programming languages, building on what students have learned in the Principles of Programming Languages course.

• Understand the major components of the compiler: Syntactic analysis, semantic analysis, code generation, and the run-time environment. Gain hands-on experience in crafting these components.
• Learn about compiler optimizations: What compilers do to generate code that is faster, shorter, and performs better. Implement many of these optimizations, and see how they improve the code.

• Be able to apply information & skills learned in the compilers course to other areas in computer science where syntactic and semantic analysis, code generation, and translation are needed.

2 Course Requirements

• 26 2-hours lectures

• 13 2-hours exercises sessions

• 4 homework assignments [15% of the grade]
  – Written problems in scanning and parsing theory
  – Programming assignments building stages of the compiler, implementing various optimizations, and various exercises in code transformation & translation. About 30 hours each, done singly or in pairs.

• Final project: Writing a code generation, and integrating the previously-written stages of the compiler into a self-contained, working compiler from Scheme to CISC assembly. About 100 hours, done singly or in pairs. [15% of the grade]: This component is mandatory

• Midterm exam [15% of the grade]

• Final exam [55% of the grade]: This component is mandatory

In this course, for a component of your final grade to be mandatory means that you shall fail the course if you fail this component. Here are some examples:

• If you fail the final exam, you shall fail the course.

• If you fail the final project, you shall fail the course.

• If you fail the midterm, but your cumulative grade in the course is 56 or higher, then you shall pass the course.
3 Detailed Syllabus

3.1 Introduction to Compiler Construction
References: 1, 3, 4, 5

- The algebraic relationship between compilation & interpretation.
- Cross-compilation, boot strapping a compiler, de-compilation.
- The stages of the compiler: What work is done in each, what kinds of errors can and cannot be detected at each, the basic algorithms that are implemented at each stage.
- Dynamic vs statically-typed languages. Early binding vs late binding. The information available to the compiler for translation, error detection, and optimizations.

3.2 Scanning & Parsing Theory
References: 1, 2, 5

- Scanner: DFA, NDFA, NDFA with $\epsilon$-transitions
- Parsing: Top-down, recursive descent parsers, parsing combinators, bottom-up parsers
- Hand-coding various parsers
- Using parser-generation tools in C & Java
- Macro expansion: Syntactic transformations, reduction to core forms in the language, variables, meta-variables and syntactic hygiene.

3.3 Programming Languages
References: 2, 3

- Functional vs Imperative programming: How change & side-effects are understood & modelled in the functional view of programming.
• Scope & its implementation: Dynamic scope (deep binding, shallow binding), lexical scope. Dynamic scope and the implementation of exception handling.

• The structure of the lexical environment, and the implications for data sharing & side effects.

• Object-oriented vs functional programming Languages. The structure of the closure compared to that of the object. Mapping of lambda-expressions to objects. The virtual method table.

• Monads & monadic programming.

3.4 Continuation-Passing Style (CPS)

References: 2, 5

• CPS as a programming technique (multiple return values, multiple continuations, co-routines, implementation of threads.

• CPS as an approach to writing a compiler: CPS, defunctionalization of the continuation, stack machine.

• CPS as an intermediate language for the compiler: Optimizations that are simpler in CPS.

3.5 Semantic Analysis

References: 3, 4, 5

• Lexical addressing, deBruijn numbering

• Identification of tail calls

• Boxing, data indirection, and motion from the stack to the heap: A comparison between quasi-functional programming languages (Scheme, LISP) and object oriented programming languages (Java).

3.6 Code Generation

References: 1, 2, 3, 4

• Layout of Scheme objects in memory. Run-time type information. Comparison with the situation in object-oriented programming languages.
• An overview of the proof of correctness of the compiler, and how it is constructed along with the code generator.

• Code generation for the various expressions in our language

• Optimization of tail calls

• The primitive procedures & support code that are provided with the compiler

3.7 The Run-Time Environment

References: 2, 3, 4

• The top level: n-LISP – value cells, function cells, property cells, etc.

• Dynamic memory management:
  – Reference counting
  – Garbage collection: mark & sweep, stop & copy, generational garbage collection

• Namespaces, modules, and their implementation

3.8 Compiler Optimizations

References: 1, 2, 3, and notes

• The tail-recursion & tail-call optimizations

• Loop optimizations & transformations

• Array optimizations

• Strength reduction optimizations

• Dead-code removal, write-after-write optimizations

• Common Sub-expression Elimination, both as a high-level and low-level optimization

• Optimizations for super-pipelined and parallel architectures
4 References


2. LISP in Small Pieces, by Christian Queinnec

3. The Anatomy of LISP, by John Allen

4. The Structure & Interpretation of Computer Programs, by Harold Abelson, et al.

5. Essentials of Programming Languages, by Daniel P. Friedman, Mitchell Wand, Christopher T. Haynes