

# CS126: Introduction to Compilers

## Missive

Fall 2003

Steven P. Reiss

### 1.0 What the Course Will Cover

This course will cover the basic concepts of writing a modern compiler. It will go over the theoretical aspects of compiler construction as well as the multitude of practical considerations. It will provide hands-on experience with writing an optimizing compiler for a modern, object-oriented language.

There are several aspects to a modern compiler. The initial concepts involve *parsing*. This is the translation of the raw input file into a form that the compiler can understand and manipulate. Parsing deals with the syntax or structure of the language being compiled. Once parsing is understood, compilers need to deal with *semantic analysis*. This involves determining what names refer to, managing types, figuring out what functions are called, and eventually translating the result of parsing into a low-level representation that is close to the machine. As high-level languages become more common and the underlying architecture becomes more complex, programmers have relied on compilers to get the best performance from the underlying machine. This is the process of *optimization*. Typically this involves taking the generated low-level representation and mapping it into an equivalent representation that will be more efficient.

Compilers have been around as long as there have been programming languages. Early compilers were messy and complex systems because little was understood about the underlying principles and everything was done by hand. As more languages were developed and more compilers were needed, people started to understand these principles and tools for compiler construction evolved. These tools are concentrated mostly around parsing. Semantic analysis, especially for modern languages, can be quite complex. While some tools have been developed here, there are no standards and the semantic analysis of most compilers is still hand-coded. Optimization has come into its own in the last fifteen years. RISC architectures and modern programming languages make optimization essential — one can't release a RISC machine without at least one optimizing compiler these days.

This course will attempt to provide an overview of all these aspects of modern compilers. The first third of the course will cover parsing and the generation of an internal representation that can be used for later semantic analysis. Here we will cover lexical and syntactic analysis as well as tools and the theories behind those tools. The second third of the course will cover semantic analysis. Here we will cover symbol tables, type algebras, expression resolution, intermediate representations, and

basic code generation. The final third of the course will cover optimization. Here we will look at the structure of an optimizing compiler as well as a number of different optimization strategies and approaches.

## 2.0 Mechanics

The course is going to be reasonably fast-paced. (Most introductory compiler courses cover optimization only rudimentally, while we will be spending about a third of the course on the subject.) There will be readings, programming assignments, occasional written assignments, and a midterm and final. The attached syllabus notes the readings, the due dates for the programming assignments, and the topics to be discussed in each class. (Note that these are all subject to change.)

The programming assignments will involve building a complete optimizing compiler for the DECAF language that generates x86 assembler code. DECAF is a subset of Java that is greatly simplified in order to make compiler writing practical in the course of a semester. Even so, however, it would be too much to ask to have you build a complete compiler from scratch in one semester. (It's about 16kloc without optimization, if done correctly.) Thus, what we are going to do is provide you with the framework of a working DECAF compiler into which you will write the essential pieces (e.g. lexical analyzer, parser, semantic analyzer, and basic code generator). We will provide the scaffolding that makes this practical and saves you a bit of grunt work (e.g. underlying data structures, intermediate representations, error handling, etc.). We have divided the programming tasks into weekly assignments that build one on top of the other. If you have severe problems with any of your earlier assignments, we will try to make an instructors version available to help you in your later work.

The readings are from the text book *Modern Compiler Implementation in Java (2nd edition)* by Appel. This text provides a readable and practical view of most of the topics of the course. The page numbers in the syllabus refer to this text. There are some additional topics that I want to cover that are not in the text. Moreover, we are going to be using a more traditional approach to the compiler than the one presented in the text and the language we are compiling is a bit more complex (and object-oriented) than the one in the text. For more theoretical and comprehensive background reading, I suggest *Modern Compiler Design* by Grune, et al. Much of the optimization implementation of the DECAF compiler and the specific algorithms we will be covering are from *Building an Optimizing Compiler* by Morgan. This is a very practical, hands-on view of an optimizing compiler, however the text is replete with errors. A more theoretical and in-depth view of optimization can be found in Muchnick's *Advanced Compiler Design and Implementation*. In addition to the text, we will be providing a few handouts to cover the particulars of DECAF and the x86 instruction set.

In addition to teaching about compilers, we hope that this course will provide you with a foundation for doing large-scale Java programming. We hope to teach, by

example as well as in class, how to structure and build a moderate-sized Java system. For this purpose, we have asked the bookstore to make available copies of the pamphlet *Elements of Java Style* by Vermeulen, et al. This book provides a set of style and coding rules (most of which we agree with) that provides a good starting point for such programming.

The midterm and final will be designed to cover the underlying concepts behind compiler construction and the several issues that we will cover but that will not be included in the implementation. The occasional written assignments will serve to point out what is important in the readings and to provide material for discussion in the subsequent class. These will be used primarily during our study of optimization since it would be unrealistic to attempt to have you actually code the optimization portions of the compiler during the one semester. (That's why we have advanced courses.)

### **3.0 Grading**

The compiler is divided up into twelve separate assignments. Each of these will count 5%. The midterm will count toward 10% of the grade and the final 20%. The written homeworks will account for an additional 10% of the grade.

Guy Eddon (geddon@cs.brown.edu) will be the TA for the course.

### **4.0 Feedback**

This is a relatively new course so that material, structure, assignments, readings, etc. have not been fully tested. We welcome your feedback throughout the course regarding any and all aspects. You can come talk to Dr. Reiss or Guy at any time or you can send your comments via email (spr@cs.brown.edu, geddon@cs.brown.edu).