Invited Editorial

Teaching the Compilers Course

Alfred V. Aho

Department of Computer Science
Columbia University
New York, New York 10027 USA
Aho@cs.columbia.edu

I've always enjoyed teaching the compilers course. Compiler design is a beautiful marriage of theory and practice -- it is one of the first major areas of systems programming for which a strong theoretical foundation has developed that is now routinely used in practice. I am a strong believer in Donald Knuth's credo: "Theory and practice are not mutually exclusive; they are intimately connected. They live together and support each other" [5].

In the past five decades, a number of forces have evolved the traditional compilers course in a computer science curriculum [6]. This article looks at how compilers courses have responded to these forces since their inception. I've organized the evolution based on material in the four books on compilers that I have coauthored and used in compilers courses over the last three decades [1-4].

Many of the earliest compilers courses put a heavy emphasis on the theory of parsing and syntax-directed translation. However, it we quickly realized that just focusing on the theory does not necessarily teach students how to build a useful compiler. This observation was positively reinforced by my interactions with the exceptionally talented researchers in the Computing Sciences Research Center at Bell Labs where I worked right after graduation. I began my career there as a computer science theorist after creating indexed grammars and nested stack automata for my PhD thesis at Princeton. With the excitement surrounding the invention of Unix, C, and C++ around me, however, I quickly became involved in the wonderful opportunities for the fruitful symbiosis of theory and practice in systems and compilers.

Many of the early tools that appeared on Unix such as diff, grep, lex, and yacc were software incarnations of algorithms that had come from computer science theory. I tried my hand at implementing some string-pattern-matching algorithms that I had devised by writing the initial versions of egrep and fgrep on Unix, and then working with Brian Kernighan and Peter Weinberger in developing the AWK programming language. Through these experiences, I learned that it often takes as much talent to implement an algorithm efficiently and correctly in practice as it does to develop it in theory. A compelling testimonial to the craftsmanship of the Unix pioneers is that many of their tools that appeared in the first versions of Unix are still in use today.

The contrast between early and modern compilers is dramatic. Early compilers were implemented with just thousands of lines of code, usually written in low-level languages like assembler or C; modern compilers come in collections often comprising millions of lines of code written in a variety of programming languages (C, C++, C#, F#, Java, and OCAML are popular current choices). Single individuals often crafted early compilers; modern compilers are typically large software engineering projects involving teams of programmers using prebuilt components and existing compiler construction frameworks.

In the early days of computer science there were relatively few programming languages; today there are thousands. In the early days of computer science, there were relatively few target machine architectures; today we still have the CISC and RISC architectures of the past, but now there are vector, VLIW, multicore, and many other eclectic architectures. The upshot of this evolution of languages and machines is that it is impossible to cover in one compilers course the diversity of algorithms and techniques used in a modern commercial compiler.

In spite of this bewildering diversity of source languages and target machines, I have found that it is still possible to offer a compilers course that gives lots of educational benefit and enormous satisfaction to students.

This is how my current compilers course has evolved. It still covers the basics of lexical analysis, parsing, semantic analysis, intermediate code generation, runtime

environments, resource management, and target code generation — tasks present in virtually every compiler. The fundamental concepts underlying these tasks are important to know because they are useful in many areas outside of compiling such as natural language processing and program verification. However, there is no time in the first compilers course to cover in any degree of depth the algorithms used in machine-independent and machine-dependent code optimization.

Implementing a compiler for a small programming language has always been a standard component of a compilers course. In the past, I used to give students a specification of the source and target languages for the compiler that they were to implement. Although this was instructive, I found students were not that excited about implementing someone else's toy language.

What students found infinitely more satisfying was to work in a small team to define their own little language and then build a compiler for it. Their eyes light up when they see programs written in their language being compiled and executed. From my industrial experience, I found that the right-weight software engineering process surrounding this kind of project made it possible for every student team to design and implement a working compiler without fail in the course of a 15-week semester.

Here's the process I used. The teams were asked to do the following tasks on the following schedule during the 15-week course, concurrently with learning the theory and practice behind compilers.

Week	Task
2	Form a team of five
4	Write a whitepaper on the proposed language modeled after the Java whitepaper
8	Write a tutorial patterned after Chapter 1 and a language reference manual patterned after Appendix A of Kernighan and Ritchie's book, *The C Programming Language*
14	Give a ten-minute presentation of the language to the class
15	Give a 30-minute working demo of the compiler to the teaching staff
15	Hand in the final project report

Once the teams are formed, each team is asked to elect a project manager, a system architect, a system integrator, a tester, and a language guru. Each of these positions carries specific team responsibilities:

- The project manager sets the project schedule, holds weekly meetings with the entire team, maintains a project log, and makes sure the project deliverables are done on time.
- The system architect defines the compiler architecture, modules, and interfaces.
- The system integrator defines the system development platform and tools, the integration environment, and a makefile to ensure the compiler components work together.
- The tester defines the test plan and test suites from the language reference manual. Each team member is expected to execute the test suites as the compiler is being developed to make sure the compiler meets the language specification.
- The language guru maintains the intellectual integrity of the language and defines a baseline system and process for managing changes to the language definition.

The TAs and I give advice as needed to the teams on how to perform these tasks. There are many benefits to this kind of project. Students get a chance to exercise their creativity in designing their own new language. They learn that writing a value proposition for a language and a detailed specification for a software system are challenging tasks. They also learn valuable skills such as project management, teamwork, and communication (both oral and written). These skills are applicable to any kind of project, not just writing a compiler. The final project report has the following chapters:

1. Introduction (written by the team)
2. Language tutorial (written by the team)
3. Language reference manual (written by the team)
4. Project plan (written by the project manager)
5. Language evolution (written by the language guru)
6. Compiler architecture (written by the system architect)
7. Development environment (written by the system integrator)
8. Test plan and test suites (developed by the system tester)
9. Conclusions containing lessons learned (written by the team)
10. Appendix containing the complete source listing of the compiler

Students get individual grades for the role they play on the team and the components of the compiler they individually write. They get team grades for the language whitepaper, tutorial, reference manual, class presentation, and project demo.

I assign a TA to mentor each team, and I usually mentor three or four teams myself. Professor Stephen A. Edwards and I have taught the programming languages and translators course with this kind of project each semester for more than five years at Columbia. There are typically 50-100 students in the class. We are always astounded by the ingenuity and novelty of the languages that the students create. They have included languages for simulating quantum computers, creating computer games, composing music, creating shared-session telecommunications services, and hosts of other innovative applications. Some of the language projects have resulted in publishable papers.

Several factors make this kind of compilers course possible. The course has as prerequisites courses in data

structures and algorithms, computer science theory, and computer systems. Students are expected to be fluent in at least Java and C. Many of the students serving as project managers have industrial experience in writing software and are familiar with design patterns such as the visitor pattern. Compiler construction tools such as lex and yacc (or their modern equivalents), integrated software development environments, and wikis facilitate distributed compiler software development so students can work independently and collaboratively on their compiler components. However, perhaps most important, the theory and practice of compiler design has matured to the point where interesting working translators can now be routinely created by beginners during a 15-week course.

The importance of the software engineering process used in this course should not be underestimated. I personally review the language whitepaper, tutorial, and reference manual carefully to make sure the compiler can be implemented with just a few thousand lines of original code. Each team member has to write at least four or five hundred lines of this code so every team member gets a taste of what it takes to write code that has to work as part of a system.

The fact that every team delivers a working compiler on schedule is actually not surprising. I tell the students to grow their language and compiler so that they only need to deliver what is working at the end of the semester. On the other hand, I ask the students to state in their language reference manual how much of their language they are committing to implementing, and what they will implement if they have extra time. In the final project demo, I ask each team to execute two programs: one of the programs chosen at random from their language reference manual and an unknown new program the TA mentor has created to test his or her team's language.

Students uniformly give glowing evaluations of the course and cite the benefits of learning enough compiler theory and tools so they can create their very own language in a semester. They frequently mention project management, teamwork, and how to give effective oral and written presentations as the most important skills learned. When the students interview for jobs, interviewers are universally impressed by the software engineering skills the students have learned from the course.

In summary, a modern compilers course can give students a satisfying opportunity to exercise their creativity. It gives them a first-hand understanding of how theory and practice can be beneficially interwoven, a small-scale scenario in which to experience good software engineering practices that can greatly improve the robustness of an implementation project, and a realistic opportunity in which to develop project management, teamwork, and communications skills that can be subsequently useful in many aspects of life.

REFERENCES

[1] Aho, A.V. and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing, Volume II: Translation*. Prentice-Hall, 1972 and 1973.

[2] Aho, A.V. and J. D. Ullman, *Principles of Compiler Design*. Addison Wesley, 1977.

[3] Aho, A.V., R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[4] Aho, A.V., M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools, Second Edition*. Pearson Addison Wesley, 2007.

[5] Knuth, D.E., *Theory and Practice*, Keynote address for the 11th World Computer Congress (Information Processing 89), San Francisco, 28 August 1989.

[6] Waite, W.M., *The Compiler Course in Today's Curriculum: Three Stategies*. SIGCSE, Houston, Texas, March 1-5, 2006, pp. 87-91.