# Cool: A Portable Project for Teaching Compiler Construction

Alexander Aiken*
EECS Department
University of California, Berkeley
http://www.cs.berkeley.edu/~aiken

## 1 Introduction

The compiler course is a fixture of undergraduate computer science education. Most CS programs offer a course on compilers that includes a substantial project where students write a compiler for a small programming language. The project often serves two distinct purposes: it teaches something about language design and compiler implementation, and it gives students the experience of building a substantial software system. A compiler project is the most complex software engineering task many students complete in an undergraduate program.

Unfortunately, developing a compiler project is labor intensive and time consuming, as it incorporates all of the problems of designing and implementing a programming language. Using a "real" language (meaning any existing programming language that has a significant number of users) does not significantly simplify the problem, because all such languages are too large to be implemented fully by undergraduates in a single course—in practice, using a real language is a choice to use a subset of a real language, which poses a substantial design problem in itself. Designing the language to be implemented is just the first problem in creating a course project, however. Precise specifications for the project must be written, any supporting software must be designed, implemented, tested, and documented, handouts must be prepared, and so on. Finally, the entire project should be implemented by course staff prior to actual use in a course, because a full implementation is the only reliable way to ensure that the project is self-consistent, complete, and tractable.

Readers who have taught or taken compiler courses may recognize that this idealized description of a compiler project sometimes bears little resemblance to reality. Time pressures can dictate that corners are cut in the design of the project, or even that the project is designed in "real time" while the course is underway—a risky strategy at best! Once an instructor has created a project, the investment made provides a strong incentive to reuse the project again and again, even beyond the point when the project becomes outdated.

Given that compiler courses are important and that compiler projects are expensive to create, it is surprising—at least to the author—that there are no standard, widely used compiler projects. Many instructors create their own projects, repeating much work that has been done many times before. Instructors do reuse projects from previous instances of the same course, but projects are not routinely shared between institutions or even between instructors at the same institution. In contrast, other areas of computer science do have widely used course projects (e.g., Tom Anderson's *nachos* project for teaching operating systems [CPA93]).

*This work was supported by an NSF NYI award.

The current situation would improve if instructors who design course projects shared the fruits of their labor more widely. This article presents *Cool*, a freely available, portable compiler project. Cool has been used for the past two years in compiler courses at Berkeley and the project is quite mature. Cool is being distributed in the hope that instructors at other institutions can benefit from the efforts of the project developers and the many students who have written Cool compilers. Cool will not suit everyone's needs or tastes; another purpose of this article is to encourage others to make compiler projects publicly available. This article concludes with a few comments about desirable features for a portable compiler project.

# 2 Cool

*Cool*, the *Classroom Object-Oriented Language*, is a small programming language for teaching the basics of compiler construction to undergraduate computer science majors. Cool is designed to be implemented by individuals or teams of two using C++ [Str91] in a Unix environment in a single semester. At Berkeley, 80-90% of the student teams complete the project each semester. The project has been designed to be relatively easy to modify, so shorter or longer projects are possible. In this paper, Cool refers both to the language and to the complete compiler project (of which the language is one part).

The Cool language is object-oriented, statically typed, and has automatic memory management. These are the essential features of a number of recent languages (e.g., Java [Jav96]). These particular features were selected because they are technically interesting, representative of a useful class of contemporary languages, and co-exist easily in one design.

The Cool project also is designed to solve three common practical problems with compiler projects. First, the project is completely modular; there are no dependencies between the assignments. In particular, a student who does a poor job on one assignment is not at a disadvantage on later assignments. Second, the project is highly portable between Unix platforms.

The third problem is that reusing projects is similar to reusing exams or problem sets. If a project has been used once at a school, the local student population develops a "memory" of the project that lasts several years, and dishonest students may submit the work of others from previous years as their own. Indeed, this problem alone may explain why so many new course projects are invented. However, making relatively modest changes to old projects reduces substantially the incentive to cheat. The Cool project is designed to be easy to modify and extend, and in fact has been substantially modified once with much less effort than would be needed to construct a new project.

The complete Cool project consists of many components beyond the language. The following subsections briefly describe aspects of the project organized around the topics of platform independence, the reference compiler, supporting software, modular assignments, formal specification, and documentation.

## 2.1 Platform Independent

Cool is highly portable and easy to install on any Unix machine with standard GNU software tools (*gmake*, *bison*, and *flex*). Cool is platform independent in two additional ways. First, Cool can be installed to support multiple architectures transparently on a common file system. At Berkeley, Cool is supported simultaneously on HP's, DECstations, and Sun workstations. A separate Linux distribution is made available for students to use on their home PC's. Second, Cool targets MIPS assembly, which can run on a simulator such as Jim Larus' *spim* [Lar]. Thus, the generated assembly code is also relatively platform independent. The Cool distribution includes *spim*.

## 2.2 Reference Compiler

The project comes with *coolc*, a Cool compiler. This reference compiler plays several roles. First, assignment skeletons and support software are extracted from the *coolc* source. Second, the phases of the compiler can be compiled separately, which supports modular assignments (see Section 2.4). Third, the reference compiler serves as a sample solution, which instructors may use to guide students. Fourth, students appreciate the ability to compare their compilers with a (hopefully) correct compiler on specific examples.

The most important function of the reference compiler is one the students never see. Implementing a reference compiler is the only reliable check that a compiler project is tractable by students in a short period of time with a minimum of drudgery. In the case of Cool, the initial implementation of *coolc* revealed several aspects of the language design that could be simplified to reduce the required implementation effort without sacrificing anything of educational value.

## 2.3 Support Software

A typical compiler uses many standard data structures (e.g., look-up tables) and has some low-level, repetitive code (e.g., templates for emitting each kind of assembly instruction). Because students have presumably had a data structures course prior to the compiler course, it is wasteful to have students implement these components. All such support code is supplied to students in the Cool project. The code is documented, and there is a separate handout describing the overall structure of the components and their interfaces.

Supplying students with support code has another advantage when the implementation language for a project is C or C++. These languages require meticulous attention to memory management details if programs are to work properly. When that care is not given, the errors are sometimes difficult to find—students can spend more time trying to find the source of a dangling pointer than learning about compilers. Providing support code that gives a moderate level of abstraction (in particular, encapsulates memory management) removes a large percentage of the possible pitfalls and enables students to focus on the most important aspects of the project.

## 2.4 Modular Assignments

Compiler projects usually are divided into four assignments: lexical analysis, parsing, semantic analysis, and code generation. The strong dependencies between these phases is an inherent and difficult problem. For example, a student who does poorly on the lexical analyzer may be indirectly penalized on the parser, because it will not be possible to thoroughly test the parser with a buggy lexer; this problem is compounded in later assignments. Conversely, without a working code generator, a student writing the semantic analyzer cannot test the code generation interface. Finally, if grading is done partly by running test cases, it is impossible to have a fair basis for grading code generation (for example) without using correct implementations of the earlier phases.

The Cool project has been structured to eliminate dependencies between assignments. The *coolc* compiler is modular with well-defined interfaces between each of the four phases. Each *coolc* phase can be compiled separately and used in any other Cool compiler that adheres to the interface. Thus, students may mix-and-match any of the components of *coolc* with any of their own components. For example, a student can construct a Cool compiler using his or her own lexical and semantic analysis and *coolc*'s parser and code generator.

### 2.5 Formal Specification

Cool comes with a language reference manual, the *CoolAid*. The *CoolAid* has two parts, an informal overview of Cool and a formal specification of the syntactic structure, static semantics, and operational semantics of Cool programs. The formal semantics is given using standard deductive type rules and structural operational semantics. A self-contained explanation of all notation is included in the manual.

It is often noted that a difficulty in teaching formal semantics is that students do not see how to apply the ideas to everyday programming problems. Arguably, no one needs formal semantics more than a compiler writer, where the language semantics serves as the compiler specification. Indeed, when told that grades are based on conformance of their compiler to the language manual, many students develop a sudden interest in formal semantics. Once acquainted with how to read the formal rules, students examine and question every detail in the course of writing the compiler. Some students report that the experience transformed their view of programming languages and formal specification. Overall, a compiler course is an excellent vehicle for introducing students to formal specifications.

### 2.6 Documentation

Besides the language manual there is a document describing all of the support code and its interfaces, a handout for each assignment, and a short *Instructor's Guide*. Documentation for all of the tools used in the project (*gmake*, *bison*, *flex*, and *spim*) is also included. Finally, all source code, including both skeleton code given to students and the reference compiler, is documented.

## 3 What Makes a Good Project?

There is a long-running and useful debate in the programming languages community about what languages and language concepts are most important to teach in undergraduate courses. In the context of compiler courses, the specific questions are: *What language should students implement?*, and *In what language should students write their compiler?*

While important, in the author's experience these two questions are not the paramount issues facing someone designing or choosing a course project. The overriding concerns are: (1) the project is well-specified and (2) the project is tractable. The problems that ensue if (1) is not satisfied, regardless of any other merits the project may have, should be clear. For (2), it is impossible for most students taking a typical course load to implement anything but a small language in a single course. Furthermore, there is little educational value in implementing the bells and whistles of realistic languages.

Given the fact that the language to be implemented must be small, the author prefers to use an invented language rather than a subset of an existing language for a compiler project. An invented language can be designed to be easy to implement rather than easy to use, which is the reverse of the usual priorities. Also, one is not constrained by existing languages and can pick and choose features from a wide spectrum. For example, Cool is object-oriented like C++, is an expression language with garbage collection like Lisp-family languages, and has a very regular syntax reminiscent of Pascal. Another advantage is that an invented language can be deliberately different from existing languages. Confronting an unknown language forces students to think consciously about the meaning of language phrases, rather than relying on intuitions borrowed from known languages.

Unfortunately, it is probably not possible to illustrate every interesting language feature in a single, coherent project. For example, Cool does not have higher-order functions, a basic feature of many modern languages. (There is no language design problem in adding higher-order functions; the problem is keeping the project small.)

Finally, the choice of language in which students write their compilers is not obvious. In part, the decision depends on whether the course also is intended to teach practical software construction or principles of language design. In courses with an emphasis on gaining experience in software construction as it is practiced in industry, the choices are C and C++. For courses with an emphasis on language design, a language such as ML [MTH90] is much better for exposing students to modern language ideas.

Cool uses C++ as the implementation language. As noted above, when students use C or C++ for implementation, it is important to give some thought to helping students minimize routine coding errors that can sap their time and enthusiasm for the project. The strategy adopted in Cool is to provide support code that offers a moderate level of abstraction for the primitive data types of the compiler, and to make a few suggestions about C++ programming style. (Specifically, it is recommended that only a small subset of C++ be used, which is the same subset used in writing the *coolc* reference compiler.) This approach has been successful; a substantial majority of students are able to complete the project within the time frame of the course.

The recent development of Java has created another candidate implementation language. Because Java is a much safer programming language than either C or C++ and because it is likely to gain popularity in industry, Java will become more attractive than either C or C++ as an implementation language for many student projects. A port of Cool to Java would be simple, as the *coolc* reference compiler is written in the Java subset of C++. The only drawback is the current lack of tools to support Java programming, particularly debuggers and a well-tested and documented Java parser generator.

# 4 Conclusions

Substantial effort has been spent developing the Cool compiler project over a two year period. It is unlikely that this experience is unique, and other educators are encouraged to spend the additional effort to make their compiler course projects available to the community for use and further development.

The Cool distribution is available from `http://www.cs.berkeley.edu/~aiken/cool`. The WWW site contains all materials a student needs to write a compiler, including the manual and all assignment handouts, source for all support code, assignment skeletons, and binaries for the phases of the reference compiler. The distribution is also available via ftp from `ftp.cs.berkeley.edu` in directory `pub/cool`. Instructors may obtain the full distribution (including source for *coolc* and test cases) by sending mail to *aiken@cs.berkeley.edu*.

# 5 Acknowledgements

The precursor of Cool is Sather164, a compiler project developed by Susan Graham and John Boyland. Sather164 is itself based on the programming language Sather [SK95]. Lok Sang Chen, Manuel Fähndrich, David Gay, Douglas Hauge, Margret Jacoby, and Carleton Miyamoto contributed substantially to the design and implementation of Cool.

Manuel Fähndrich, Susan Graham, and Margret Jacoby provided comments on earlier drafts of this paper.

# References

[CPA93] W. Christopher, S. Procter, and T. Anderson. The Nachos instructional operating system. In *1993 Winter USENIX Conference*, pages 479–488, January 1993.

[Jav96] The Java Tutorial: Object-Oriented Programming for the Internet. http://java.sun.com/java.sun.com/tutorial/intro.html, 1996.

[Lar] J. Larus. Spim. http://www.cs.wisc.edu/~larus/spim.html.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[SK95] D. Stoutamire and M. Kennel. Sather revisited: A high-performance free alternative to C++. *Computers in Physics*, 9(5):519–524, September 1995.

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991.