Cool Instructor's Guide

Alex Aiken
EECS Department
University of California, Berkeley
aiken@cs.berkeley.edu

Cool, the *Classroom Object-Oriented Language*, is a small programming language designed for teaching the basics of compiler construction to undergraduate computer science majors. I have used Cool for the past two years in teaching the compiler course at Berkeley. In the process, I've learned that developing a polished compiler course project is a labor-intensive exercise in software development. Having complete that exercise, I am distributing Cool in the hope that instructors at other institutions can benefit from the effort and experience of myself, the other developers of Cool, and the many Berkeley undergraduates who have written Cool compilers and provided criticism of the project.

Cool is designed to be implemented by a team of 2 or 3 undergraduates in C++ in a single semester. At Berkeley, 80-90% of the student teams complete the project each semester. Interesting subsets of the language can be implemented by individuals in a single quarter, and even shorter projects are possible.

This Guide is divided into five sections covering installation of the system, an overview of the contents of the distribution, a brief rationale of the Cool design, an outline of how I have used Cool as a course project, and a discussion of possible alternative ways to use Cool.

1 Installation

Cool is being distributed both in binary and source form. Binary installation only requires retrieving the `.u` file for a particular architecture from `http://www.cs.berkeley.edu/~aiken/cool` and following the directions in `README.INSTALL`. Source installation instructions are in the file `INSTALL` in the top-level directory. The system is designed to be very portable and builds with no changes on many systems. The system has been built successfully without modification on Sun, HP, and DEC Alpha, and DECstation workstations, as well as under Linux on PC's (both pre- and post-ELF).

The Cool WWW site has only binary distributions available. I am happy to send the source distribution to instructors, but as I am still using Cool at Berkeley, I am not making source code for a Cool compiler publicly available.

2 Overview

The Cool binary distribution comes with:

- `CoolAid`, a reference manual for Cool. The manual describes the syntax and semantics of Cool, as well as how to execute Cool programs. The manual is divided into an informal and a formal part. The first, informal part gives an overview of the language, examples, and an English description of each of the language's constructs. The second, formal part of the manual gives precise descriptions of the lexical structure, grammar, type rules, and operational semantics of Cool.

- The `Cool Tour`, an overview of the source code provided to students to assust them in building a compiler.
- Documentation for GNU tools students use to write portions of the project: `flex`, `bison`, and `gmake`.
- `coolc`, a reference Cool compiler. `coolc` has been extensively tested over the last two years; there are currently no known bugs in the compiler.
- The lexer, parser, semantic analysis, and code generation components of `coolc`. The Cool compiler project is modular and students may, for example, combine their lexer and semantic analysis with `coolc`'s parser and code generator to produce a complete compiler.
- `spim`, a MIPS simulator distributed by James Larus. `coolc` generates MIPS assembly code.
- A suite of example Cool programs, small and large.
- A set of programming assignments, including both handouts and code skeletons.

The source distribution also includes source code for `coolc`, a a large collection of valid and invalid Cool programs useful for testing Cool compilers, and Makefiles for installing the system.

3 Cool

Cool is a small language, designed for teaching. Cool is object-oriented, strongly typed, and has automatic memory management. These are the essential features of a number of languages, of which the most recent and probably best known is Java.

Cool syntax is, for the most part, simple and regular. Cool deliberately does not "look" exactly like any language with which students are likely to be familiar. The intention is to encourage students to think consciously about the semantics of Cool when implementing their compilers, rather than relying entirely on intuitions borrowed from other languages.

The lexical structure of Cool is quite straightforward, with the exception of a few twists added intentionally to make the lexical analysis assignment more challenging (e.g., nested comments).

As part of the project, students must write a grammar for Cool. The grammar given in the manual is a partial solution, but it uses a number of non-context-free features. Almost all Cool constructs except infix arithmetic expressions have both opening and closing keywords (e.g., `begin ...end`). This design makes for verbose Cool programs, but it also makes designing, debugging, and extending the parser significantly easier. The collection of infix expressions introduce ambiguities which students must resolve either by writing a LALR(1) grammar or using precedence declarations.

Cool has a fairly conventional object-oriented type system, in which (single) inheritance also defines the subtyping relation. The only subtle feature in the Cool type system is `SELF_TYPE`, the type of the `self` object. Cool has several different kinds of identifiers with varying scope rules illustrating common design choices.

The execution of Cool programs uses a standard object-oriented model with dynamic dispatch. All memory management is automatic; `coolc` has a generational garbage collector. A couple of surprises have been engineered into the operational semantics. In particular, the `case` expression provides a form of runtime dynamic type checking that is trickier to implement correctly than it first appears. Also, the semantics of `new` turns out to be more involved than students imagine.

4 Teaching Cool

As with most compiler projects, a Cool implementation is most naturally built in four stages: lexer, parser, semantic analysis, and finally code generation. I add a fifth assignment: writing a Cool program. Writing a program in Cool before writing a compiler familiarizes students with the language, the manual, `coolc`, and provides the students with a substantial test case for their project. For a 15-week semester course, the time I allot for each assignment is:

	assignment	*weeks*
1	Cool program	1
2	lexical analysis	2
3	parser	2.5
4	semantic analysis	4
5	code generation	4

A very important aspect of the project is that students are able to proceed even if they do not complete one of the assignments. Students can use any of the four components of `coolc` to test the component they are writing. Thus, a student writing a parser may test it by linking with `coolc`'s semantic analysis and code generation and either the student's or `coolc`'s lexical analyzer. Thus, there are no dependencies between the assignments that penalize a student who fails to finish one of the early components.

Most students find lexical analysis to be the easiest assignment. However, in my experience it is also the assignment with the lowest average grade, as it is very easy to make obscure errors using `flex`-like tools.

The parsing assignment requires the least amount of coding effort. The assignment still requires a considerable amount of time, however, because students must learn to use a parser generator and the details of Cool's representation of abstract syntax trees.

There is a large step up in implementation complexity between the parsing assignment and the semantic analysis assignment; where the parser can be written in at most a few hundred lines, the semantic analyzer generally requires 1000-1500 lines of code. It is important to stress to the students that this assignment has more time allocated to it for a reason, but, inevitably, a few students lulled into a false sense of security by earlier assignments fail to complete this assignment. I make the fourth and fifth assignments easier by telling students not to worry about memory management—they should just allocate as much space as they like and not worry about deallocating objects that are no longer in use. I also emphasize correctness as the most important goal. Students are free to use simple, inefficient implementations so long as the compiler works as specified.

The final assignment is a little larger than the semantic analysis component, but many students find it easier overall because there is no error checking (the hard part of semantic analysis is making sure that all erroneous programs are detected) and because they can compare their compiler's generated code directly against that of `coolc`.

Each of the assignments comes with a handout (LaTeXsource is available in the `cool/handouts` directory) and a `README` file (see the directories under `cool/assignments`). The instructions should be modified with information for individual courses (e.g., how to turn in assignments).

5 Alternatives

There are a number of ways that the *Cool* project can be varied to save time and effort and still be a coherent project. These variations inlcude

- *Subset the language.* There are two orthogonal ways to subset the language. Removing a language feature will make all phases of the compiler easier. For example, dropping `case` expressions simplifies all phases, but particularly semantic analysis and code generation. Particular phases can also be simplified. For example, dropping `SELF_TYPE` makes semantic analysis significantly easier to implement.
- *Eliminate a phase.* Semantic analysis or code generation could be skipped and the corresponding component from `coolc` always used in its place.
- *Give bigger skeletons.* The given assignment skeletons are modest; more `coolc` source code could be given as a starting point.

Another alternative is to use Java as the implementation language for the semantic analysis and code generation assignments. It should be very easy to translate the *coolc* source and skeleton code into Java, and the fact that Java is a safe programming language would make the project easier for many students. When development tools (such as debuggers and a good parser generator) are available for Java, a port of the entire system to Java would be worthwhile.

Of course, it is also possible (though not likely) to have too much time in a course, in which case the project can be extended. I usually assign an extra-credit optimization phase (the students implement any optimization of their choice and measure the results) in the last week of the course.