

Using a Class on Compiler Design to Teach Software Construction

(Draft Paper, Sept 2000)

Thomas Gross
Departement Informatik
ETH Zuerich
8092 Zuerich, Switzerland

Abstract

A class on compiler design is offered by many departments because it allows a student to see the interplay between theory (finite state machines, grammars, formal languages) and practice (language translation, problematic features of modern programming languages). Yet this purpose, although important, is not the only reason to include a compiler class in the curriculum. This paper reports how a compiler class can also provide the base to teach other subjects that students should be exposed to: software reuse, re-engineering, experience with patterns and software architectures, and reverse engineering. These topics are important even if the student never writes a language translator later on in his or her career.

1 Introduction

There exist a number of well-known reasons to include a class on compiler design in the course catalogue, and compiler design is often part of the list of required classes.

- Compilers are a good example of the interplay between theory and practice. Most students have encountered formal languages (and the Chomsky hierarchy), finite state and push-down automata, and various graph algorithms sometime in an earlier class. The compiler class allows the student to apply these concepts, e.g., in parser and lexer generators, or when building an abstract syntax tree. And well-known real languages provide ample illustration of the pitfalls of language design: nested comments that cannot be handled by a scanner based on a finite state machine, a macro facility that invites programming errors, ambiguity in the syntactic definition, etc.
- A good understanding of compilers is essential to appreciate modern computer architectures.
- Compilers provide a good playground to explore how to decompose a system into modules.

And since every computer scientist uses at one time or another some language translator, every computer scientist should know how a compiler works.

But in addition to these “classical” reasons, a class on compiler design can also be used as a vehicle to teach other concepts. Computer science students should learn about software reuse; since most software development is done in the context of extending or maintaining an existing system, it is important for students to develop experience with reusing software. Students should learn about software architectures and learn to identify and use patterns [2]. And students need experience with re-engineering (and maybe even reverse engineering).

These concepts are difficult to convey in lectures alone; they need to be grounded in real experience. But time constraints prohibit that the student participates in a real re-engineering project, or participates in the maintenance of a real software system. Since a compiler is often the first “complete” software system that a student encounters (is exposed to the sources), the compiler class provides an interesting vehicle to learn (by doing and by listening) software reuse, re-engineering, and software architectures and patterns. These topics are not novel; a number of the compiler (or other) courses that are taught today have included them. In this paper we present one specific organization of the compiler course that maximizes the student’s learning experience with regard to these topics. These ideas are based on three iterations of teaching a compiler class at 2 institutions.

2 Course description

The compiler class that is the topic of this paper is a 1-semester or 1-quarter course of 12–14 weeks. Typically, the participants are juniors and seniors, and this is their first exposure to the topic of compiler design.

Our underlying compiler model is fairly conventional: a compiler consists of a front-end that includes a scanner and parser (this part constructs a syntax tree), a semantic checker that processes the syntax tree to produce an intermediate representation (IR) of the program, and a code generator that maps the IR nodes to assembly

instructions. If the compiler includes an optimizer, it works with the intermediate representation produced by the semantic checker.

2.1 Input language

The source language for the compiler is a simplified subset of the intersection of C and Pascal. In its base version, the language contains one basic type (Integer) and arrays with compile-time bounds, conditionals and while-loops, as well as procedure calls with reference and value parameters. Depending on the students' interest or progress (or as extra assignments for extra credit), the language can be extended to include Boolean variables and/or structs (records), to ease the restriction of compile-time bounds for arrays (i.e., to allow a procedure to accept array parameters with arbitrary bounds), or to include also a for-loop. Further extensions (e.g., to include pointers, to interface to a runtime system with garbage collection, to allow separate compilation) are possible, but given the wealth of material in the core class, these extensions probably have to wait for a follow-on class.

2.2 Implementation language

Over the course of the semester, the students have to implement a compiler for this simple language. The compiler is implemented in Java. There are two reasons for the choice of implementation language: Java is a lot saner to read than C++, and there are reasonable tools (parser generator, lexer generator) available. However, there are almost no dependences on the implementation language as long as there is a reasonable lexer and parser generator.

2.3 Target machine(s)

The student is given the freedom to choose as a target (for the compiler's output) a real machine or a simulator. The simulator is a simple instruction level simulator of a basic RISC machine (the DLX machine[3]), but since the language does not require the use of byte instructions, even that simulator can be simplified. About 60%–80% of the students in the past have used the simulator as the target.

The only constraint on the choice of a real machine is that the machine supports registers; this condition rules out the byte code of the JVM. Dealing with machine resources is an important aspect of any code generator, and the use of a stack machine would deprive the students of a significant part of the learning experience. The majority of those who chose a real machine picked the SPARC machine. (Other machines the students have used include the Intel IA32, the DEC/Compaq Alpha, and the SGI MIPS architecture). The preference for the SPARC is probably motivated more by easy access than by any specific machine feature.

Week	Topic
1	Structure of a simple compiler Intermediate representations I
2	Intro to code generation for straight-line code Linearization
3	Scanning, lexical analysis Context free grammars, top-down parsing
4	Context free grammars, bottom-up parsing
5	Table-driven bottom-up parsing
6	Parsing tools: CUP and JLex SLR parsing
7	Intermediate program representation II Symboltable, semantic checker
8	Intermediate program representation III Control flow constructs Code generation for function bodies
9	Function calls Parameter passing, calling conventions
10	Control flow analysis Global data flow analysis I
11	Global data flow analysis II
12	Reaching definitions Live variables
13	Examples for use of GDF information Uninitialized variables
14	Spare

Figure 1: Course outline.

1	Code generator for basic blocks
2	Lexer and parser
3	Code generator for a function
4	Code generator with function calls
5	Simple global data flow analysis and program analysis

Figure 2: Assignments.

2.4 Content

Figure 1 shows the compiler topics that are covered each week. There are five assignments (see Figure 2) that reinforce the contents of the course. These assignments provide the vehicle to explore reuse, re-engineering, and software patterns.

Backend I After an overview of the overall structure of a compiler, the class starts with code generation for a sequence of expressions. In the input language that is accepted by the compiler that is implemented by the students during the semester, assignment statements are expressions, and expressions do not allow control transfers (nor is there a ‘?’ operator like in C). Therefore a sequence of expressions maps into a straight-line sequence of assembly operations.

Code generation for straight-line code is simple but far from trivial. Our compiler works with expression trees,

so the code generator has to decide which subtree to evaluate first. If we want to minimize the memory instructions, the result of subtree that is evaluated first should be kept in a register, so the larger subtree should be evaluated first.

The students taking this class have some prior exposure to computer architecture, but nevertheless, the step from understanding the execution of an instruction to selecting an instruction sequence for a source-language operation is not easy. However, since the students use as targets well-known architectures, there exists a version of the GNU C compiler (`gcc`) for the target machine, and we have the opportunity to encourage the student to perform reverse engineering. We use a few simple C programs (and then compile those with the `-S` flag, which leaves the assembly language output of the compiler in a file, usually with the extension `.s`) to demonstrate how this compiler deals with integer addition, multiplication, etc.

The first assignment is the implementation of a code generator for straight-line code, to match the topics of the first 2 weeks of lectures. The intermediate representation format is already defined, so the main task is to map a small list of IR nodes (arithmetic operations, assignment and use of variables) to machine instructions. The students are given a compiler skeleton that produces instruction trees for expression sequences (including assignments). To simplify the assignment, the student's compiler does not have to compile an expression if it requires more registers than are supported by the machine. (Since the students can pick a machine of their choice, there is no way to determine this number of registers a priori.)

The first assignment challenges the students in two dimensions. First, the student has to master a subset of the target machine to produce code (see above). Second, the student's compiler must produce an output file that can be assembled by an assembler (either the machine specific assembler that is invoked by `gcc` or an assembler for the simulator). Since the input language includes a simple input operation (`read(x)` reads an integer from `stdin` and assigns the value to `x`) and a simple output operation, students have to implement those as well. (Without input/output a smart student may "optimize" every sequence of expressions to a single `nop`.) The cheapest way (in terms of engineering time) to implement those I/O statements is to use the C library functions `printf` and `scanf`. The linkage to these functions can be explored again by using the `gcc` compiler.

The same technique applies to getting the execution environment right: compile an empty C program

```
int main() {  
}
```

and then paste the code generated by the compiler into

the generated assembly (`.s`) file. After this exercise in reverse engineering the students have overcome any reluctance to explore their target computers.

The simple code generator provides also an excellent vehicle to illustrate the usage of design patterns. In almost all possible designs for the code generator, there is an opportunity to point to the visitor, strategy, and iterator patterns, and to either the abstract factory or the factory method pattern.

Front end The next topic is scanning and parsing, but in contrast to other compiler classes, we spend less than 1/3 of the total class on this subject. The idea is to explain the students the principles of a scanner and parser generator and then let them use a specific suite. `CUP` and `JLex` are in our experience reasonable systems [4].

The matching assignment asks the students to use these tools to generate a syntax tree for the full language (not just the expression subset that we used in the first assignment). One immediate problem is that the parser generator must be combined with the symbol table management routines (which have to be developed by the students), and this task can be addressed as a re-engineering problem. The facade pattern can be used for this design problem to unify the symbol table.

The semantic checker is treated in conjunction with the front end and provides another reason to re-engineer the parser generator (semantic checks require to combine syntactic information, information kept in the symbol-table, and the rules of the language).

Backend II The emphasis on backend (code generation) issues reflects our experience that this part of the compiler is the most challenging. Since this is the part that requires the students to learn about resource management and resource conflicts, we put more emphasis on the backend than on the frontend.

After semantic checking, the class discusses the issues of code generation that go beyond assignment 1, i.e. the treatment of conditionals, loops, and function calls. These constructs are common to almost all popular programming languages. The range of different machine architectures also provides an opportunity to expose to the students the many different approaches that could be taken by a compiler.

There are two assignments that reinforce these topics. First, the students are asked to enhance their compiler to translate a single function. Then the second assignment adds function calls (and the associated machinery for parameter passing). To complete this assignment, the students have to integrate their expression code generator (from assignment 1) with the parser+semantic checker. This task immediately provides a need for re-engineering, since the solution for the first assignment must be extended to deal with a set of full-fledged sym-

bol table management routines. Furthermore, expressions can occur anywhere (e.g., in a conditional stmt or a while-loop). Once function calls are implemented, the compiler must be able to manage the frame for a procedure (so that it can construct a parameter list and allocate storage for temporaries). This requirement makes it possible to compile expressions with arbitrary register requirements. Whereas the simple code generator of assignment 1 was allowed to determine that an expression required too many registers (for this specific machine architecture), the compiler at the end of assignment 3 must be able to compile arbitrary expressions. This widening of the requirements (although announced to the students early on) nevertheless requires usually some adjustment to the solution from assignment 1.

Also, depending on the design of the parser and semantic checker, a student may have to re-engineer those modules to allow extension to a code generator. However, even conventional compiler classes that proceed from scanner to assembler provide an opportunity for this kind of re-engineering.

(It is also possible to combine these two assignments into one single, longer-running assignment. Although a more complex assignment gives the students a chance to sharpen their skills in design and modularization, this strategy is not without risk since some students may put off the assignment until it is too late. However a working compiler for the core language is a necessary precondition to appreciate the following part on global flow analysis, so current practice is to divide the code generator into two assignments.)

Global flow analysis and optimization The last third of the class (approximately) is devoted to global flow analysis and optimization. Again, the overriding concern is to expose the student to the core material and to provide an opportunity to apply the key concepts in a simple compiler. Since the input language is fairly simple, control flow analysis is not an issue and can be dealt with only briefly, so that most of the time is available to discuss global data flow.

The last assignment requires the implementation of a simple iterative solver for forward data flow problems (extra credit is given to those that also implement a solution for backward flow problems). A global data flow analyzer can be compact [1] and the students are encouraged to use a design that minimizes their effort. (Since the end of the semester is approaching rapidly the time the assignment is given out, most students do not need extra motivation to implement a compact solution.) The resulting optimizer is trivial from a compiler design perspective (function calls and arrays immediately cause worst-case assumptions) but the students nevertheless pick up the principal idea that some dynamic program properties (e.g., that a variable is not initialized on all paths to a use) can be computed ahead of time by the compiler.

This assignment is more complex than the earlier ones. A reasonable solution may be based on more than one module; one module to collect the information and another one to work with the information. It may make sense to divide this last assignment into two parts: first implement the iterative solution to the global dataflow problem, and then use this information in a subsequent assignment to detect un-initialized variables. Another option is to include some mid checkpoint to ensure that everyone is making continuous progress.

3 Reflection

Each assignment (except the first) provides an opportunity to reuse either the result of a previous assignment or a substantial building block developed by someone else (parser generator, scanner generator). A “sequential” compiler class (i.e., a class that starts with the front end and then extends this compiler assignment by assignment) provides also some opportunity for software reuse (since the result of the previous assignment is extended). However, the reuse demanded by the “inverted” organization (the code generator comes first) is different. The student has to integrate a substantial piece of his or her software into the compiler and has to add significant extensions (use of expressions in arbitrary statements, storing of temporaries in the case of a register shortage). The important aspect is not the reuse but the student’s experience: each reuse requires some form of re-engineering and is never totally free.

The use of patterns provides a good starting point for a discussion of software architectures. Compilers are well-understood software systems, so there exist a number of proven organizations. In a real compiler, the choice of the IR has a significant influence on the compiler structure. Here, the IR is specified by the instructor (so that the class can be jump-started with assignment 1). Nevertheless, even in this environment, there exist a number of different organizations. (A one-pass design is possible for the first 4 assignments, although a multi-pass design allows for an easier modularization. If a student chooses a one-pass design, the optimizer of the last assignment can be made to work on the assembly language output with little effort.)

3.1 Compromises

Input language This class makes compromises (relative to other compiler classes) with regard to front-end issues. The input language is extremely simple (but powerful enough to program many sample programs). And the class does not cover any syntax error correction or error reporting techniques that go beyond what is supported by the parser generator. Although the cryptic message `Syntax error` may still be the state-of-the-art with some commercial compilers, it reflects a poor human-computer interface.

Parser generator Another compromise (to have time for global optimization later on) is to cover only the principles of parser generation. The parser generator used for the assignment (CUP) is actually for a class of languages that is not covered in detail in the lectures. In the lecture, we discuss parsing of SLR grammars (with the associated construction of the tables for the parser) and then briefly discuss how this idea can be extended to other language classes, i.e. LR(k) and LALR(k) grammars. The parser generator works for LALR grammars (and this is indeed the most popular technique today, since *efficient* parsers for such languages can be generated). However experience shows that after students understand the principle of a parser generator for an SLR grammar, the extension to other language classes is easy to grasp. Our objective is *not* to teach the student how to implement a parser generator but how to *use* it; a student should understand the principles but does not have to handle all the details.

3.2 Optimization

When presenting the algorithms to collect and use global data flow information, a number of textbooks are unnecessarily complex. If a compiler uses basic blocks, subsequently the data flow module collects information for basic blocks. However then the optimization algorithms have to translate the block-level information into information that is valid at the point in the program where an optimization is applied. E.g., the global phase will discover the set of definitions that reaches the start of a basic block, but local definitions reach all points between the definition and the end of the basic block.

It is easy to circumvent this complexity by never introducing basic blocks. Such a decision may increase the running time of the iterative solution (since now more nodes have to be visited), but this increase is irrelevant for a student compiler. Basic blocks are an optimization (and can be introduced to the students at the end of the class as such), but at least one well-known commercial optimizing compiler also did not use basic blocks for major parts of its optimizer [5].

In one instance of teaching this class, global dataflow was only given cursory treatment. In retrospect this was unfortunate since the concept of (static) flow analysis and its use by an optimizing compiler are very important. As a consequence, the material on local code generation was reduced, and only simple optimizations are discussed. A student who wants to work on a real compiler must have an in-depth understanding of optimizations. This can be obtained in another class (and this one would probably employ SSA as a representation and not employ an iterative solution to the dataflow problem [6]). However for those students that do not follow this direction, a simple optimizer is sufficient to appreciate the ideas.

4 Conclusions

This class demands a lot from the students. In addition to learning the principles of compiler design, a student also picks up some substantial programming experience. This experience reinforces earlier exposure to classes that emphasize “programming-in-the-small”. By providing a good place to illustrate the use of patterns, this class also builds a bridge to the difficult topic referred to as “programming-in-the-large”, i.e. how to structure a large(r) software system. By starting the assignments with part of the backend, we create extra opportunities for the student to explore software reuse and to learn about its difficulties.

Acknowledgements

Peter Lee was the first to teach a compiler class that started with code generation. Matteo Corti, Angela Demke, and Syed Shah helped as teaching assistants. The source language is based on a language designed by Nicklas Wirth.

References

- [1] Adl-Tabatabai, A., Gross, T., and Lueh, G. Code reuse in an optimizing compiler. In *Proc. OOP-SLA '96* (October 1996), ACM, pp. 51–68.
- [2] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns, Elements of Object-Oriented Software*. Addison Wesley, 1995.
- [3] Hennessy, J. L., and Patterson, D. A. *Computer Architecture A Quantitative Approach (2nd Edition)*. Morgan Kaufman, 1995.
- [4] Hudson, S. et. al. Cup parser generator for java. <http://www.cs.princeton.edu/appel/modern/java/CUP/>, 1999.
- [5] Lowney, P. G., Freudenberger, S. M., Karzes, T. J., Lichtenstein, W. D., Nix, R. P., O'Donnell, J., and Ruttenberg, J. C. The multiframe trace scheduling compiler. *Journal of Supercomputing* 7, 1,2 (March 1993), 51–142.
- [6] Muchnick, S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.