Compiler Construction: Craftsmanship or Engineering?

William M. Waite

William.Waite@Colorado.EDU
University of Colorado, Boulder, CO 80309-0425, USA

Abstract. Engineering is defined as the application of scientific principles to practical purposes, as the design, construction and operation of efficient and economical structures, equipment and systems. Computer science is concerned with efficient and economical systems, but what are the "scientific principles" that we apply in their design, construction and operation? Compiler construction was one of the first areas of computer science to be treated formally, and is often used as a touchstone for application of scientific principles in our field, but does formalization imply scientific principles? The issues of craftsmanship and engineering in compiler construction are bound up with the set of problems that compilers must solve and the ways in which people solve problems by computer; our positions on these issues determine our approach to compiler research.

Keywords: Problem solving, formal methods, complexity, modularity, reusability

1 Introduction

Two conferences, held under the sponsorship of the NATO Science Committee in 1968 [20] and 1969 [6], introduced the term "software engineering". This phrase "was deliberately chosen to be provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering [20]."

What does software development in general and compiler construction in particular have in common with "established branches of engineering"? Our products are not manufactured in the same way as buildings and motors. Our work is based on theoretical foundations, but they involve discrete rather than continuous mathematics. We are not limited by the physical properties of matter, although space and time certainly bound what we can do. In order to see whether the analogy is useful, we need to look at it more carefully.

My dictionary defines engineering as "the application of scientific principles to practical purposes, as the design, construction and operation of efficient and economical structures, equipment and systems." We need to design, construct and operate efficient and economical systems, but what scientific principles are we applying?

Philosophers customarily divide the sciences into two main groups on the basis of the way scientists arrive at conclusions. Mathematicians justify their conclusions on the basis of deduction from some set of given axioms. Physicists justify their conclusions as generalizations of behavior observed in nature. Thus mathematics is classified as a *deductive* science and physics as an *inductive* science.

Civil and mechanical engineers apply principles from inductive sciences like physics and chemistry to create useful products. Those principles provide a framework within which the engineer operates, both guiding and constraining the engineer's efforts. Software engineers apply principles from deductive sciences that guide their efforts but do not significantly constrain them.

Section 2 summarizes some properties of deductive principles, and Sect. 3 indicates how such principles are applied in compiler development. Thus it appears that our behavior satisfies the definition of engineering, but Sect. 4 argues that we probably need to move more in that direction.

2 Scientific Principles for Compiler Construction

A system S is a nonempty set D (or possibly several such sets) of objects among which certain relationships exist [16]. When the objects of the system are known only through the relationships of the system, the system is *abstract*.

A *lattice* [26] is an example of an abstract system. The relation of the system is $\leq$, and for any $a, b, c \in D$ the following hold:

- $a \leq a$
- if $a \leq b$ and $b \leq c$ then $a \leq c$
- if $a \leq b$ and $b \leq a$ then a and b are identical

A lower bound of $X \subseteq D$ is an element a such that $a \leq x$ for all $x \in X$. The greatest lower bound of X is a lower bound c of X such that $a \leq c$ for every lower bound a of X. Upper bounds and the least upper bound are defined analogously. Every pair of elements must have a greatest lower bound and a least upper bound if the system is to be a lattice.

Only the *structure* of an abstract system is established by its definition. What the objects are, in any respects other than the way in which they fit into the structure, is left unspecified. Any further specification of what the objects are yields a *representation* of the abstract system. The representation is a system whose objects satisfy the relationships of the abstract system but have some further properties as well.

The type system of a programming language is usually a representation of a lattice [12]: Each object is a type. For any types t_1 and t_2, $t_1 \leq t_2$ if and only if values of type t_1 are coercible to values of type t_2. (One value is coercible to another if the compiler is allowed to insert a type conversion operation without explicit programmer input.)

Another representation of a lattice is the system of addressing modes used by machine instructions to obtain operands: Each object is an addressing mode, and for any modes m_1 and m_2 $m_1 \leq m_2$ if and only if operands specified by mode m_1 can also be specified by mode m_2. For example, an operand specified by a literal addressing mode can also be specified by a memory addressing mode or (after loading the value into a register) by a register addressing mode.

A computer program is inevitably an implementation of an abstract system. The computer cannot perform manipulations on the basis of representations, because it has no understanding of the world at large. Thus the task that we undertake when we write a program is to develop an abstract system that captures exactly the relationships of interest in solving the given problem. That task involves four distinct kinds of activity:

- *Understanding* the problem
- *Pondering* the problem to obtain an idea for a solution
- *Reasoning* about the idea to show that it solves the problem
- *Implementing* the idea in a programming language

Of course these activities do not proceed sequentially; there is much iteration as certain aspects of the idea turn out to be wrong, or inefficient, or irrelevant. Nevertheless, we can identify all of the individual things we do with one or another of these activities.

The term "pondering" is due to Dijkstra, who said that the purpose of pondering is to reduce the amount and complexity of the reasoning needed to show that the idea solves the problem:

> The ability to "ponder" successfully is absolutely vital. When we encounter a "brilliant, elegant solution", it strikes us, because the argument, in its austere simplicity, is so shatteringly convincing. And don't think that such a vein of gold was struck by pure luck: the man who found the conclusive argument was someone who knew how to ponder well [8].

Both understanding and pondering often involve making connections with other problems and methods. In any discipline, we make progress by increasing the number of problems with known solutions [23]. Computing is not an exception: Studies have shown that the only reliable discriminator between novice and expert programmers is the number of problem and solution patterns that they can access [13].

The scientific principles of software development (and of compiler construction in particular) are therefore useful abstract systems and the deductions that can be made within them. For example, computability theory and computational complexity involve deductions within abstract systems based upon simple relationships among states and operations. They provide the only "physical limitations" on computer algorithms. They give us ways of deciding what is and is not possible, and of characterizing the behavior of an algorithm on the basis of problem size. As long as real computers are representations of those systems, the deductions will be applicable to all software we develop.

Formal languages are the abstract systems most frequently associated with compiler construction. Any interesting programming language contains an infinite number of strings, and formal language theory allows us to give finite descriptions of that infinite set. We can use the description to reason about the language, deducing such properties as expressiveness and ambiguity.

The semantics of a programming language construct are often described in terms of the semantics of the components of that construct. For example, the meaning of a conditional is described in terms of the meanings of the condition, then-part and else-part. A tree is an abstract system on whose elements the "component of" relation exists, and therefore it is also associated with compiler construction.

There are many other abstract systems that are useful in compiler construction, but I shall make no attempt to list them here because the purpose of this section is simply to characterize the nature of the scientific principles on which the question of engineering vs. craftsmanship is based.

3 Application of Scientific Principles

If we agree that the scientific principles of compiler construction constitute a set of useful abstract systems, then the application of those principles to practical purposes is the use of those abstract systems in building a compiler. For a given project, some of the abstract systems used will exist at the start and others developed during the course of the project may later come into general use as new scientific principles. Still others will be highly dependent on the particular problem and will never be seen again.

In the terms of software engineering, application of existing abstract systems by the compiler writer is *reusing* them. Initially applied to simple incorporation of code fragments, this term has more recently been applied to a wide variety of artifacts [17,21]. Here we will be concerned with only three kinds of reuse:

Code The artifact to be reused is specific source code. It may exist as a module in a library, or as a fragment of an existing program.

Generator The artifact to be reused is a solution to a class of problems that has been embodied in a tool.

Design The artifact to be reused is an explanation of how a class of problems can be solved.

Reuse is only possible when, during either the understanding or the pondering activity, we recognize the problem at hand as an instance of some problem class for which a solution is known.

A problem class is characterized by some *requirements space* that distinguishes one instance of the class from another. Code reuse requires that the developer understand only this requirements space, and be able to select the appropriate code on the basis of the portion of the requirements space that it covers. For example, a developer might select a dynamic storage manager on the basis of whether or not it provided garbage collection facilities [5].

Generator reuse requires that the developer not only understand the requirements space, but also understand how to describe a problem to the generator. For example, an attribute grammar is a formal language capable of describing the structure of a tree and relating computations to that structure. Generators that produce efficient programs to carry out the the computations described by an attribute grammar exist, but to use them the developer must understand the attribute grammar language in addition to the requirements space of tree computations.

Design reuse involves reading and understanding the appropriate literature, and then implementing the design in a manner compatible with the particular problem instance being solved.

Our ability to apply scientific principles in compiler construction rests with the availability of code fragments, generators and literature. Most of these artifacts deal with *tactics*: techniques for solving single problems that arise in the course of writing a compiler. When we try to build a compiler using these tactics, we find that very careful selection is required if they are to work smoothly together. Taken as a group, they must implement a coherent compilation *strategy*.

For example, one of the simplest strategies is the classical one-pass approach taken in many current compiler classes. If the source language is suitably defined, a program can be checked for adherence to the language definition in a single pass over the text without retaining a representation of the program in memory. Often the entire translation can be accomplished as the source program is being checked, although the quality of the generated code may leave something to be desired. The result is a fast compiler with a relatively simple structure [2,3,4].

If source language properties require that the compiler retain a representation of a portion of the program in memory for semantic analysis, then code generation tactics should be chosen to take advantage of that requirement. Thus selection of tactics for solving one subproblem of the compilation problem will depend on the characteristics of another subproblem.

4 Research Agenda

Software engineers concerned with reuse believe that the most significant reuse products involve specifications:

> Specification reuse, which offers the highest payoff of all, is a form of generative reuse [21].
> By focusing on a narrow domain, the code expansion in application generators can be one or more orders of magnitude greater than the code expansion in programming language compilers [17].

Compiler construction is the enabling technology for these forms of reuse. An *application generator* is a form of compiler that accepts a specification in a domain-specific language and automatically selects algorithms and data structures so that the developer can concentrate on *what* the system should do rather than *how* it is done.

Figure 1 is a summary of the characteristics of application generators, taken from Krueger's paper on software reuse [17]. It makes a strong case for the importance of application generators.

Abstraction Abstractions come directly from the application domain. These high-level abstractions are mapped directly into executable source code by the generator.

Selection Application generator libraries have not received much attention in the literature. The parallel between software schemas and application generators suggests, however, that library techniques could be used to select among a collection of application generators.

Specialization Application generators are specialized by writing an input specification for the generator. Due to the diversity in application domain abstractions, the techniques used for specialization are also widely varied. Examples include grammars, regular expressions, finite-state machines, graphical languages, templates, interactive dialog, problem-solving methods and constraints.

Integration Application generators do not require integration techniques when a single executable system is generated. In cases in which a collection of generators produce a collection of subsystems, composition is best done in terms of domain abstractions.

Pros Since high-level abstractions from an application domain are automatically mapped into executable software systems, most of the conventional software development life cycle is automated. This significantly reduces cognitive distance.

Cons Because of limited availability of application generators, many of which have narrow domain coverage, it is often difficult or impossible to find an application generator for a particular software development problem. It is difficult to build an application generator with appropriate functionality and performance for a broad range of applications.

Fig. 1. Reuse in Application Generators

The drawbacks of application generators presented in Fig. 1 are closely related. Availability is limited because they are expensive to build, and the only incentive to broaden the range of application is to spread the cost. Nevertheless, it is often cost-effective to build an application generator to generate one software system [19], and becomes more so the cheaper the application generator. Since application generators always involve specification languages, anything that reduces the cost of implementing processors for specification languages will reduce the cost of the application generator.

A specification language processor is a compiler, so in order to reduce the cost of building an application generator we need to reduce the cost of building a compiler. According to the software engineers, the best way to reduce the cost of building a compiler is to use an application generator whose application domain is compiler construction! This should not be surprising, because scanner and parser generators have been part of the compiler writer's toolbox for years [14,18]. Compare the specification of a simple expression language using these tools to early papers on expression analysis [24] to get an idea of the leverage that an application generator can provide.

Scanning and parsing only account for about 9% of the time and 11% of the code in a typical compiler, so to continue to increase our leverage we need to go to application generators that deal with larger subproblems. A key point is to begin to embody strategy as well as tactics in the generator.

We already have experience with application generators for the scanning [10,18], parsing [7,14], tree computation [15,28], and code generation [1,22] subtasks of a compiler. Several such generators have been combined under the control of an expert system to create an application generator for complete compilers [11,27]. An evaluation of such generators shows that the code they generate runs as fast as hand code, but uses more memory [25]. Additional research is needed to improve space efficiency and broaden coverage.

Creation of a more comprehensive application generator for the compiler construction domain is really just a process of making our understanding of the compilation process explicit. None of the details are left to the imagination, as they usually are in a reusable design. Many of those details involve things that everyone supposedly understands, but that are easy to do poorly. For example, the speed of a generated compiler and a hand-coded compiler were recently compared by using each to process a test suite of 471 programs [25]. The generated compiler was about 5.4% faster on average. Careful analysis of the compilation time revealed that the source text input routine was responsible for a significant part of speed differential: A carefully optimized routine was produced by the generator, but the hand coder had simply used the C library.

Complete compiler generators give tremendous leverage to the compiler expert. In order to fulfill their promise of lowering the cost of other application generators, however, they must also make the abstractions that constitute our scientific principles available to people with limited experience. That means packaging support for developing a processor design as part of the generator [9], and providing training materials covering basic compiler construction technology.

Even for the expert, it is not sufficient to have only the compiler generator. Input specifications that specialize (Fig. 1) it to analyze common programming languages and generate code for common machines are required as well. Such specifications would make it possible for (say) a person interested in optimization research to quickly and cheaply generate a program to build an appropriate representation of the code to be optimized. If they want to make extensions to the source language to convey additional information those changes can be made in specifications rather than in code. Thus the researcher obtains the necessary infrastructure cheaply and can get on with the interesting aspects of their work.

5 Conclusion

Compiler construction as a discipline can be considered engineering according to the definition given in Sect. 1. There is a set of scientific principles, and those principles are applied to practical purposes. Design reuse is practiced widely, and some generators are used. Code is also reused in specific cases.

Craftsmanship is by no means unknown, however. Optimizing compilers are usually built by craftsmen on an engineered base, and compilers for new or unusual languages involve ad-hoc solutions.

Compiler construction is an enabling technology for application generators, and in order to support this area we need to provide higher levels of automation. Such improvements would also reduce the cost of entry for compiler researchers who wish to investigate problems involving specific compiler components.

References

1. Aho, A. V., Ganapathi, M. & Tjiang, S. W. K., "Code Generation Using Tree Pattern Matching and Dynamic Programming," *ACM Transactions on Programming Languages and Systems* 11 (October 1989), 491–516.
2. Ammann, U., "The Method of Structured Programming Applied to the Development of a Compiler," in *Proceedings of the International Computing Symposium 1973*, North-Holland, Amsterdam, 1974, 94–99.
3. Ammann, U., "Die Entwicklung eines PASCAL-Compilers nach der Methode des Strukturierten Programmierens," Eidgenössischen Technischen Hochschule Zürich, Ph.D. Thesis, Zürich, 1975.
4. Ammann, U., "On Code Generation in a PASCAL Compiler," *Software - Practice & Experience* 7 (1977), 391–423.
5. Boehm, H-J. & Weiser, M., "Garbage Collection in an Uncooperative Environment," *Software - Practice & Experience* 18 (September 1988), 807–820.
6. Buxton, J. N. & Randell, B., eds., *Software Engineering Techniques*, NATO Science Committee, April 1970.
7. Dencker, P., Dürre, K. & Heuft, J., "Optimization of Parser Tables for Portable Compilers," *ACM Transactions on Programming Languages and Systems* 6 (October 1984), 546–572.
8. Dijkstra, E. W., *On the Teaching of Programming, i.e. On the Teaching of Thinking*, International Summer School on Language Hierarchies and Interfaces, Munich, 1975.
9. Fischer, G. & Nakakoji, K., "Empowering Designers with Integrated Design Environments," in *Artificial Intelligence in Design '91*, J. Gero, ed., Butterworth-Heinemann Ltd., Oxford, 1991, 191–209.
10. Gray, R. W., "A Generator for Lexical Analyzers That Programmers Can Use," *Proceedings USENIX Conference* (June 1988).
11. Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. M. & Waite, W. M., "Eli: A Complete, Flexible Compiler Construction System," *Communications of the ACM* 35 (February 1992), 121–131.
12. Hext, J. B., "Compile-Time Type Matching," *The Computer Journal* 9 (February 1967), 365–369.
13. Jeffries, R., Turner, A. T., Polson, P. G. & Atwood, M. E., "The Processes Involved in Software Design," in *Acquisition of Cognitive Skills*, J. R. Anderson, ed., Lawrence Erlbaum Associates, Hillsdale, NJ, 1981, 254–284.
14. Johnson, S. C., "Yacc - Yet Another Compiler-Compiler," Bell Telephone Laboratories, Computer Science Technical Report 32, Murray Hill, NJ, July 1975.

15. Kastens, U., "LIGA: A Language Independent Generator for Attribute Evaluators," Universität-GH Paderborn, Bericht der Reihe Informatik Nr. 63, Paderborn, FRG, 1989.

16. Kleene, S. C., *Introduction to Metamathematics*, D. Van Nostrand Company, NYC, 1952.

17. Krueger, C. W., "Software Reuse," *ACM Computing Surveys* 24 (June 1992), 131–184.

18. Lesk, M. E., "LEX - A Lexical Analyzer Generator," Bell Telephone Laboratories, Computing Science Technical Report 39, Murray Hill, NJ, 1975.

19. Levy, L. S., "A Metaprogramming Method and its Economic Justification," *IEEE Transactions on Software Engineering* SE-12 (February 1986), 272–277.

20. Naur, P. & Randell, B., eds., *Software Engineering*, NATO Science Committee, January 1969.

21. Prieto-Díaz, R'en, "Status Report: Software Reusability," *IEEE Software* 10 (May 1993), 61–66.

22. Proebsting, T. A., "Simple and Efficient BURS Table Generation," *SIGPLAN Notices* 27 (July 1992), 331–340.

23. Shaw, M., "Larger Scale Systems Require Higher-Level Abstractions," in *Proceedings Fifth INTL Workshop on Software Specification and Design*, IEEE Computer Society, 1989, 143–146.

24. Sheridan, P. B., "The FORTRAN Arithmetic-Compiler of the IBM FORTRAN Automatic Coding System," *Communications of the ACM* 2 (February 1959), 9–.

25. Sloane, A. M., "An Evaluation of an Automatically Generated Compiler," *ACM Transactions on Programming Languages and Systems* 17 (September 1995), 691–703.

26. Stone, H. S., *Discrete Mathematical Structures and Their Applications*, Science Research Associates, Chicago, 1973.

27. Waite, W. M., Heuring, V. P. & Kastens, U., "Configuration Control in Compiler Construction," in *Proceedings of the International Workshop on Software Version and Configuration Control*, Teubner, Stuttgart, FRG, 1988.

28. Zimmermann, E., Kastens, U. & Hutt, B., *GAG: A Practical Compiler Generator*, Lecture Notes in Computer Science #141, Springer Verlag, Heidelberg, 1982.