The Compiler Course in Today's Curriculum: Three Strategies [*]

William M. Waite
University of Colorado
Boulder, CO 80309
William.Waite@Colorado.edu

ABSTRACT

The broadening of computer science education has called into question the roles of many traditional core courses. In order to remain viable, courses such as compiler construction must provide a coherent view of their subject matter that fits with the rest of the institution's curriculum. Three strategies have evolved for this course. As described in this paper, each strategy provides a model that a professor can use to design an appropriate course for their situation.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computers and Information Science Education—*computer science education*

General Terms

Design

Keywords

Curriculum issues, course pedagogy, compilers, software engineering, theory, tools

1. INTRODUCTION

ACM's Curriculum 2001 defines a course called "Programming Language Translation" (CS240s) [14]:

> Introduces the theory and practice of programming language translation. Topics include compiler design, lexical analysis, parsing, symbol tables, declaration and storage management, code generation, and optimization techniques.

Unfortunately, it is impossible to cover all of this material, in depth, in a single course. That fact was recognized as early as 1983, when a curriculum study at Carnegie-Mellon University concluded that the traditional "comparative languages" and "compiler" courses should be re-packaged as a three-course sequence [19]. This proposal was not widely accepted, and with the pressure placed on the systems area by accreditation requirements [7] there is little chance of acceptance in the future.

If we are to continue offering a course in this area, we need a viable strategy that will allow us to make consistent decisions about viewpoint, depth of coverage, and detailed tactics resulting in a coherent view of the subject. Course design must take the overall goals of the curriculum and the prerequisites into account, providing the necessary bridges between old and new material. Over the years, experience has shown that there are at least three candidate strategies:

1. *Software project*: Emphasize design patterns, teamwork, and programming methodology by constructing a compiler to meet assigned specifications.
2. *Application of theory*: Emphasize the role of theory to enable automation of compiler tasks, and illustrate the limitations of that theory.
3. *Support for communicating with a computer*: Emphasize the broad applicability of compiler technology to implement languages for special purposes.

In the next three sections I shall expand on each of these strategies, indicating how they can be used as decision-making frameworks for course design. Each section ends with a statement of what the students should have learned as a result of taking such a course. That statement serves as a concrete characterization of the course goals.

2. SOFTWARE PROJECT

Compiler construction can be taught as a software project course. This strategy is closest to the traditional one of "have the students write a compiler", but the focus is on teaching software development techniques rather than compiler algorithms. Because of this focus, the specific design and implementation methodology must be chosen by the instructor. An object-oriented approach will simplify the task, and probably provides the most utility to the students in today's environment.

Languages: The source language that the student compilers must accept and the target language that they must

[*]This work was supported by the National Science Foundation under grant EIA 008625. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

produce should also be specified by the instructor. Having all of the compilers carrying out the same task is advantageous because it encourages the entire class to talk among themselves, comparing their understanding of the issues and sharing solutions to common problems. It also allows the instructor to present specific compilation techniques when appropriate, without distracting the students from the main topics of software development.

I suggest that the source language be object-oriented, giving the students some insight into the principles of such languages and the reasons for restrictions that they might have found onerous in their own programming. Cool [2] and MiniJava [3] are two possibilities. My experience has been that it is best to select a language designed by someone other than the instructor; students tend to believe that languages designed by their instructors are not "real".

I also suggest that the target be either the Java Virtual Machine (JVM) [17] or the .NET Common Interface Language (CIL) [11]. Both of these targets hide a number of details that complicate the compiler without giving the student any new insights into either compiler construction or software development. Also, implementations of both target languages are available and therefore the students (and the instructor) can execute compiled programs and verify that they behave correctly.

Re-use: Re-use is an important tactic in software development, and this course should emphasize re-use on various levels [15]. At the highest level there is the *software architecture*, which encapsulates problem solutions in a particular domain (e.g. compiler construction) by a large-grain framework. Our understanding of the software architecture, component design, and implementation of a compiler has developed over the last 50 years and is described well in a number of textbooks. Unfortunately, after a cursory explanation of the overall structure, all of these treatments effectively walk the reader through a fairly specific compilation task. Most devote significant space to discussion of formal language theory that underlies parsing, a slant more appropriate to the strategy discussed in Section 3. Thus it seems best not to select a standard compiler text for this course, but to put a selection of these books on reserve for the students to consult when they need more information on the architecture of a compiler.

Generators encapsulate the process of constructing software components from problem specifications [6]. Scanner- and parser-generators are widely available, and allow software developers to re-use sophisticated algorithms without the need to understand those algorithms in detail or to spend time reproducing them. Using one or more generators to produce the structural analysis component of the compiler allows the students to begin getting feedback from their compiler much earlier than they would have if they had coded the structural analyzer by hand. It also illustrates this form of re-use, making both the costs and the benefits explicit (see Chapter 2 of [21] for more on this aspect).

A *design pattern* encapsulates methods for a general class of interactions [10]. The structural analyzer produces a tree according to a slightly modified version of the "Composite" design pattern (a number of abstract classes, representing tree nodes with similar semantics, play the "composite" role). The "Flyweight" design pattern is used to represent entities (like variables) whose state is independent of the tree nodes that reference them, and the "Visitor" design pattern is used to separate the structure of the tree from the computations that take place upon it. Most compiler texts don't make specific use of these design patterns, but rather develop ad-hoc solutions. Thus it is important to provide the students with explicit instruction in the use of design patterns if they have not encountered them elsewhere in the curriculum.

Code encapsulates an algorithm for a specific problem. There are many situations in which the compiler writer must solve a well-known problem such as reading text, converting strings to numbers, or creating a hash table. Solutions to these problems are usually available in libraries, and there is no point in re-creating them. Students should be encouraged to make use of Java class libraries, the C++ Standard Template Library, and other libraries available to them. One of the goals of any implementation course is to increase the student's store of "canned solutions", thereby moving them along the trajectory from novice to expert [13].

Methodology: Extreme programming is well matched to the needs of an academic software project because of its emphasis on small groups and rapid feedback, and textbooks that help students to understand this methodology are available [21]. Nevertheless, it is important for the instructor to realize that group work is anathema to many students in computer science and that specific interventions are necessary to get students to collaborate effectively [23].

One of the basic practices of extreme programming is "test first" [4]. It requires a developer to write a test for a feature *before* writing the code to implement that feature. The current version of the program is then run, and of course the test fails. After implementing the feature, the test succeeds. All tests are retained for regression testing, to provide evidence that changes haven't broken existing facilities. This strategy is very important in compiler development, where we need both "conformance" tests (programs that the compiler should accept) and "deviance" tests (programs that the compiler should reject). These test programs actually help to explain the desired behavior to both the implementor and the user. The professor should initially supply the class with a few conformance and deviance tests, and ask the students to submit additional tests with every milestone. Students should be encouraged to exchange tests, and to develop a database of tests that they can run automatically.

Students who complete a course designed according to this strategy will be familiar with the process of developing software in a team environment and with providing appropriate quality assurance for that software. They will understand the roles of tools, design patterns, and class libraries in the re-use of solutions to component problems. They will be able to decompose compiler design problems into appropriate lexical, syntactic, and semantic analysis components and to construct useful transformations of the resulting decorated abstract syntax tree to an intermediate representation.

3. APPLYING THEORY

Ad-hoc implementations of compiler tasks often contain subtle errors. The erroneous behavior is usually triggered by unforeseen combinations of linguistic elements. This problem was recognized early on, and a large body of theory has been developed for (or adapted to) design of correct compilation algorithms.

Theory derives its power from its ability to construct a consistent model of a domain, and to make guarantees about the correctness of operations in that domain. Unfortunately, languages used by humans are often not particularly consistent. We usually introduce inconsistencies for brevity, to avoid having to specify "obvious" things, in order to make problem statements easier for us to understand.

The combination of a need for theory and a need to escape from it makes compiler construction an excellent medium to explore the uses and limitations of theory in Computer Science. In a curriculum that emphasizes theory, it provides a way to both validate the material in other courses and show how adjustments must be made in practice. For example, properties of the members of the Chomsky hierarchy [5] would probably be familiar to students in such a curriculum. The compiler construction course would then show why we use regular languages to describe basic symbols and comments, while context-free languages are appropriate for describing the phrase structure above that level.

Formal Languages: Scanner and parser generators [1] are appropriate tools for such a course, because they allow the student to explore the limitations of the theory without significant programming: The instructor can select features from existing languages that cannot be specified formally using those tools, and the class can consider whether such cases are best handled by extending the theory, escaping from the tool, or prohibiting the feature.

Most programming languages are context-sensitive, but practical issues lead us away from use of context-sensitive grammars. Instead, we use tree computations to bind identifiers, determine types for expressions, and check context conditions. These computations effectively establish relationships among data items associated with tree nodes. Once the desired relationships have been determined, implementation requires a tree traversal strategy and a mechanism for storing the data items.

Attribute Grammars: An attribute grammar [18] is a formal system that allows a user to define data items (attributes) and the relationships to be established among them. A tool can then be used to generate the entire evaluation algorithm and storage mechanism from that definition.

Maintenance and extension of a compiler usually requires changes in the set of tree computations. If the compiler was written by hand, these changes are often needlessly complicated because they must fit into the existing tree-traversal strategy. A designer using an attribute-grammar-based tool is not constrained by an existing evaluation algorithm, since the tool will generate a new one that is optimal for the altered computations.

The theory behind such tools is complex and interesting, and the tools themselves are compilers that translate a formal specification language into an efficient evaluation algorithm (usually a few percent better than a hand-coded algorithm [20]). Since attribute grammar theory may very well not have been taught earlier in the curriculum, the compiler course provides an opportunity to show the students how the essence of a particular problem (construction of an evaluation algorithm for tree computations) can be described abstractly (by a set of dependence graphs) and then solved by formal manipulation of that abstraction. This gives them insight into the way theoreticians bring general concepts to bear on practical problems, and how they obtain results that are of practical use. Although these points could be illustrated by scanner and parser generators, the axiomatic nature of the presentation often obscures them in a formal languages course; here they can form a central focus of the development.

Tree Automata: If a typical RISC machine [16] is chosen as the target for a translator project, there is a significant conceptual mismatch between source and target. This mismatch is handled by using a computation over the decorated abstract syntax tree of the source program to generate an intermediate representation similar to the JVM or CIL code discussed in Section 2. That intermediate representation is then analyzed, and an abstract target program built. Computations over the abstract target program decorate it with information about target resources like registers and memory locations. The final code is produced by a computation over the decorated abstract target program.

We can preserve important structural information present in the abstract source program by using a tree as the intermediate representation. Just as we obtain the structure of the source program by parsing a string, we obtain the structure of the target program by parsing this intermediate tree. The theory of automata on infinite trees [22] tells us how to use a tree grammar to describe the rich structure of the abstract target program by taking advantage of contextual relationships in the intermediate tree. A tree parser generated from that tree grammar [9] makes all of the necessary decisions, avoiding the need for the user to explicitly test for specific tree fragment contexts.

The tree parser verifies that the intermediate tree is correct according to the tree grammar that we have supplied. Unlike the situation with the parser for the original text, however, any error is due to a fault in the compiler. How can we guarantee that the "front end" of our compiler will never produce an incorrect intermediate tree? Again, the theory of automata on infinite trees can be applied to construct a tool that tells us whether our tree grammar covers all of the intermediate trees that the analyzer can construct [8].

Students who complete a course designed according to this strategy will be familiar with the strengths and weaknesses of various theoretical models. They will understand why theory is important in assuring the quality of software, and why one must be judicious in one's selection from several possible theoretical solutions for a given problem. They will be able to apply a variety of theories to compilation problems, and to give advice on structural issues to designers of programming languages.

4. SUPPORTING COMMUNICATION

In order to get help with a problem (from either a human or a computer), you need to describe your understanding of that problem to your potential helper. The easier it is to effectively describe your understanding to that helper, the more valuable they are to you. Therefore one way to make a computer more valuable is to improve its ability to correctly interpret problem statements made by its users. Compiler technology enables us to support the interpretation of problem statements written in languages similar to those used by humans to communicate with one another, thus making the computer more useful.

Although very few (if any) of our students will write conventional compilers, most will have the occasion to develop a text processor of some kind. It might be a program to make consistency checks in some data description, to format programs in some language, or to generate a set of linked web pages from a description of the desired content. All of these tasks can be solved quickly and easily using existing compiler construction tools. The compiler construction course can familiarize the students with such tools and their application to a wide variety of text processing problems.

Language Analysis: Many students are ill-equipped to think about the process of communicating with a computer because they have never really thought about the process of communicating with a person. Thus it is appropriate to begin with the role of structure in human communication. For example, consider the classic example

> Time flies like an arrow.
> Fruit flies like a banana.

These two sentences *look* very similar and yet most people would argue that they have very different meanings. We can express the difference in meaning by diagramming each to make the differing syntactic structures visible [12]. It is possible to use *either* syntactic structure for *either* sentence, but once the syntactic structure has been stated for a sentence then the meaning of that sentence is fixed.

A human reader determines an appropriate meaning in this example from their understanding of the words, their impression of the sentence as a whole, and the context in which it occurs. It is only afterwards that they can formalize that meaning by a diagram. Computers, on the other hand, do not yet have the capacity to determine the meaning of a sentence in this way. When we communicate with a computer, therefore, we need to use a language that is limited by the analysis techniques that the computer has available.

The standard compiler architecture (see Section 2) effectively characterizes the available analysis techniques. It is not necessary to have a deep understanding of the way compiler components operate in order to construct an effective model of the process. The important thing is to understand how these components apply in a wide variety of situations. Thus the course should introduce a comprehensive compiler construction kit [1] for the students to use in exploring language constructs. It should focus on how to gain leverage from using generators to produce code from specification languages [6].

Abstract Representation: Trees should be the only syntactic structures considered in this course. The main reason is that a tree expresses the "component" relationship commonly used to describe semantics. We see this in natural languages (describing the semantics of a sentence in terms of its subject and predicate) as well as in programming languages (describing the semantics of a *while* statement in terms of its expression and controlled statement). Another reason, however, is that there are powerful tools to support tree computations. As discussed in Section 3, complete implementations of tree walkers can be generated from specifications of the computations they must carry out.

An object-oriented conceptual model of tree computation closely fits the best specification languages, and therefore it is worthwhile to spend some time discussing how simple tree computations are carried out in an object-oriented language. The "Visitor" design pattern [10] is especially apropos, since it helps to explain how values are conveyed from one context to another and how storage for those values can be separated from the tree itself.

Tool Use: The assignments in a course using this strategy should be a series of small projects, aimed at showing the variety of situations in which domain-specific languages and generators created by compiler construction kits can provide leverage to the practicing programmer. If possible, students should be encouraged to try to develop languages that describe recurring problems in other courses and build generators to simplify their lives in those courses. In any case, the instructor must explicitly point out that they are gaining leverage by using the tools in the compiler construction kit [24].

Tool use is an especially contentious point with many students. They perceive the effort devoted to learning how to use the tool as wasted, and they believe that the tool unfairly inhibits their creativity. They argue that such tools "are never used in industry" and thus that they are being deprived of the chance to learn how compilers are "really built". It is important to be able to counter these arguments with consistent reasoning based on an overall strategy for the course, because otherwise the students will view the material simply as a set of arbitrary and irrelevant requirements.

Students who complete a course designed according to this strategy will be able to use tools to create robust processors for linguistic descriptions of problem understanding. They will be able to develop languages for specific problem domains, including design of appropriate syntax and relevant semantics. They will understand the decomposition of a text-processing problem into components whose characteristics can be described formally, and will be familiar with the corresponding notations and tools.

5. SUMMARY

We have examined three strategies for teaching a compiler course. In each case, we can make a statement about generally-useful skills that a student will be able to take away from the course. That statement helps the students to understand why they should study the material being taught, and how they should approach that material. It provides a framework that allows them to direct their efforts and to link the concepts and exercises to previous knowledge.

The classical approach of "have the students write a compiler" has none of these benefits. Students who believe that they will never write a compiler don't have a clear picture of why they should study this material. A complete compiler, especially one coded by hand, is a large program for students at the undergraduate and first-year graduate level. Moreover, that program has three major tasks (analysis, transformation, and resource allocation) with very different characteristics. The result is that what the students carry away with them is the memory of late-night coding marathons and a mish-mash of techniques for solving very specific problems posed by a particular source language and target machine.

Each strategy described in this paper also provides the professor with a rationale for making decisions about the selection of material to cover, assignments to give, and tools to use. Table 1 summarizes possible choices that we have discussed in Sections 2-4. It should be emphasized that these are examples that have been found to work in practice. The important point, however, is that the professor should make the choices based on the learning objectives, and be able to explain to the students why those choices were made.

Table 1: Characteristics of the Strategies

Strategy	Project	Theory	Communication
Goals	Apply software engineering techniques	Relate theory to practice	Learn to build domain-specific generators
Material	Compiler decomposition Re-use of solutions Agile methodology	Formal language applications Attribute grammar applications Tree automata applications	Language analysis Abstract representation Tool use
Assignments	Compiler project	Tools based on theory	Application generators
Tools	Scanner/parser generator Design patterns Class libraries	Scanner/parser generator	Compiler toolkit

I have used all of the strategies discussed in this paper, as well as the classical approach, in 40 years of teaching compiler construction in a variety of environments. My experience tells me that the classical approach is no longer viable, but that any of the three strategies presented here can be very successful. Whichever you use, however, you need to be consistent in your decisions and communicate your rationale clearly to your students. In this way you provide them with the meta-information they need in order to fit the material you are presenting into their evolving model of the discipline.

6. REFERENCES

[1] Catalog of compiler construction tools. http://catalog.compilertools.net/.

[2] A. Aiken. Cool: The classroom object-oriented language. http://www.cs.berkeley.edu/˜aiken/cool/.

[3] A. W. Appel and J. Palsberg. *Modern compiler implementation in Java*. Cambridge University Press, Cambridge, UK, second edition, 2002.

[4] K. Beck. *Test-Driven Devlopment*. Addison-Wesley, Reading, MA, 2003.

[5] N. Chomsky. Three models of the description of language. *IRE Transactions on Information Theory*, 1:113–124, 1956.

[6] J. C. Cleaveland. Building application generators. *IEEE Software*, 5(4):25–33, July 1988.

[7] Computing Accreditation Commission. *Criteria for Accrediting Computing Programs*. ABET Inc., Baltimore, MD, 2004. http://www.abet.org/forms.shtml.

[8] H. Emmelmann. Testing completeness of code selector specifications. In U. Kastens and P. Pfahler, editors, *Compiler construction: 4th International Conference CC'92*, volume 641 of *Lecture Notes in Computer Science*, pages 163–175, 1992.

[9] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG: Fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, Apr. 1992.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[11] J. Gough. *Compiling for the .NET Common Language Runtime (CLR)*. Prentice-Hall, Upper Saddle River, NJ, 2002.

[12] H. C. House and S. E. Harman. *Descriptive English Grammar*. Prentice-Hall, New York, second edition, 1950. http://webster.commnet.edu/grammar /diagrams/diagrams.htm.

[13] R. Jeffries, A. T. Turner, P. G. Polson, and M. E. Atwood. The processes involved in software design. In J. R. Anderson, editor, *Cognitive Skills and their Acquisition*, pages 254–284. Lawrence Erlbaum Associates, Hillsdale, NJ, 1981.

[14] Joint Task Force on Computing Curricula. Computing Curricula 2001 — Computer Science. Final report, IEEE Computer Society, Association for Computing Machinery, Dec. 2001. http://www.computer.org/portal /cms_docs_ieeecs/ieeecs/education/cc2001/cc2001.pdf.

[15] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[16] J. Larus. SPIM: A MIPS32 simulator. http://www.cs.wisc.edu/˜larus/spim.html.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Company, Reading , MA , USA, second edition, 1999.

[18] J. Paakki. Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[19] M. Shaw, S. Brookes, M. Donner, J. Driscoll, M. Mauldin, R. Pausch, B. Scherlis, and A. Spector. Proposal for an Undergraduate Computer Science Curriculum for the 1980's, Part II: Detailed Course Descriptions. Technical Report CMU-CS-83-157, Carnegie-Mellon University, 1983.

[20] A. M. Sloane. An evaluation of an automatically generated compiler. *ACM Transactions on Programming Languages and Systems*, 17(5):691–703, Sept. 1995.

[21] D. H. Steinberg and D. W. Palmer. *Extreme Software Engineering*. Prentice-Hall, Upper Saddle River, NJ, 2004.

[22] W. Thomas. Automata on infinite trees. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 165–170. MIT Press, 1990.

[23] W. M. Waite, M. H. Jackson, A. Diwan, and P. M. Leonardi. Student culture vs group work in computer science. In *Proceedings of the 35th ACM Technical Symposium on Computer Science Education*, pages 12–16, New York, 2004. ACM Press.

[24] D. E. Yessick and J. Jones. Reinventing the wheel or not yet another compiler compiler compiler. In *Southeast ACM Conference*, 2002. http://citeseer.ist.psu.edu/705273.html.