A Modern Approach to Teaching Computer Translation

Manuel E. Bermudez, *University of Florida, Gainesville, FL, USA*

***Abstract*—In the Computer Science curriculum, the compilers course is dying. It has been relegated to a "specialized, optional" role in the ACM 2001 curriculum. However, some fundamental topics are covered only in that course, such as syntax analysis, computer translation with applications outside compilation, and language specification mechanisms such as regular expressions. In this paper, we present the outline of a completely reorganized and modernized course on computer translation. The approach resembles a spiral: topics are discussed repeatedly and in increasing depth, accompanied by implementation projects that illustrate them. We abandon the traditional goal of implementing a production compiler, as well as other topics that are too compilation-specific. Applications are discussed in areas such as data mining and software engineering.**

***Index Terms*—compilers, translation, Computer Science curriculum.**

I. Introduction

A little more than a decade ago, in Computer Science curricula at major universities around the world, a course in compiler construction was considered indispensable to the formation of the undergraduate student. It was the rare Computer Science academic program that did not have a required compiler construction course in its curriculum. It was considered "embarrassing" to have a graduate of one's university, who was ignorant of the issues and techniques taught in compiler construction courses. This situation has now changed. For the last decade or so, the compilers course in the most Computer Science curricula has been in decline.

In this paper we discuss the reasons for this decline, and the reasons we believe reform is needed. Then we present the outline of a completely reorganized and modern approach to the teaching of the topic of Computer Translation. Our approach is intended to rescue from oblivion topics that we still believe are fundamental to the formation of fledgling computer scientists, but that risk falling by the wayside as a consequence of the demise of the compilers course. At the same time, we cast away many topics that are too compiler specific, and whose continued presence (and emphasis) in compiler courses contribute to the problem.

The remainder of this paper is organized as follows. In Section II we discuss the particulars of the decline of the compilers course. In Section III we discuss the current set of problems that beset compiler courses everywhere, and justify our assertion that reform is in order. In Section IV we present our version of the modern Computer Translation course, and its novel organization that resembles a spiral. We discuss the advantages of our approach. In Section V we discuss applications of our approach, outside of the area of compilation. Finally, in Section VI, we present conclusions.

Author's address: Computer and Information Science and Engineering Department, University of Florida, E301 CSE Building, P.O. Box 116120, Gainesville, FL 32611-6120, USA. Author's email: manuel@cise.ufl.edu. Author's website: www.cise.ufl.edu/~manuel This work has been supported in part by a gift from the Microsoft Corporation.

II. The Decline of the Compiler Course

No one questions that the compilers course is less central to the Computer Science curriculum today, than it was some years ago. There are many reasons for this.

First, there is the maturity of the compiler discipline itself. Compiler construction techniques have evolved at a much slower pace in recent years, and Computer Science curricula tend to give higher importance to emerging technologies. The traditional compilers course was a great opportunity for exposing the student to a wide variety of data structures: trees, symbol tables, graphs, etc., but those topics are now considered sufficiently covered in basic courses such as Data Structures.

Another reason for the decline is the proliferation of languages and paradigms in recent years, prompting CS departments, curriculum experts, and textbook writers to focus their efforts on issues of design and implementation of programming languages, rather than the implementation techniques traditionally covered in a compilers course. As a result, CS curricula today are much more likely to require a Programming Languages course, than a Compilers course.

Yet another reason is that CS curricula are more aware now that few CS graduates will make a living either writing or maintaining compilers, and the skill-set required is considered highly specialized. Similarly, fewer faculty make compilers their main area of research, and as new, exciting topics in Computer Science have made their appearance, such as computer networking and issues related to the WWW, the number of faculty willing to teach the compilers course has diminished.

Finally, there is the matter of trends in CS Curriculum. During discussions leading to the 2001 ACM Computer

Science Curriculum, Professor Mary Shaw of Carnegie-Mellon University said, "Let's organize our courses around ideas rather than around artifacts … Engineering schools don't teach boiler design – they teach thermodynamics. Yet two of the mainstay software courses – compiler construction and operating systems – are system-artifact dinosaurs" [1]. The 2001 ACM-recommended curriculum addressed this problem by relegating the compilers course to the status of "advanced/supplemental" (i.e. elective) material.

III. Reforming the Compilers Course

The previous discussion regarding the demise of the compilers course can be summarized in one phrase: times change. Indeed, whenever times change, there are winners and losers; there would seem to be little room in the CS undergraduate curriculum of the future for a compilers course. However, some of the problems that have made the compilers course less popular in recent years are due to the organization of the course material itself. We now enumerate some of them.

- Most often, the compilers course is oriented towards the implementation of a compiler. The traditional goal of making it an exercise in writing "real" compiler is part of the reason for the course being considered too specialized, and no longer fundamental in the curriculum.
- The course project often conflicts with the topic sequence in the course. The project usually consists of implementing a compiler from scratch, perhaps with the aid of tools such as `lex [2]` and `yacc [3]`. Students implement the compiler as the term progresses, developing the various compiler components as the relevant topics are covered in the course. There is often a mismatch between the material coverage in the textbook (and in class) and the project. To design and implement one compiler phase, it is often necessary to understand how its design will affect later phases, which have not yet been covered in the course. Since the size of the project is prohibitive for a single student, instructors typically assign teams of students to develop one or more compiler components. This results in individual students being exposed to only one part of the compiler, or only one part of the language being implemented. The student misses out on the "big picture" of the entire translation process, until perhaps the very end of the course. For that student, the course is like a mystery novel: the outcome (and plot) is not revealed until the very end. In addition, if the compiler is implemented from scratch, the student does not see actual results until the very end of the course. If the implementation effort falls short, or is plagued with last-minute problems, the student often winds never having a fully working compiler. Because of this, the compilers course is often perceived by students as being "esoteric", "difficult", and "frustrating". All of this tends to reduce its popularity.
- The classical textbook on Compilers by Aho, Sethi and Ullman [4], known as the "Red Dragon" book, is not pedagogically well-suited for an undergraduate course. It a large, comprehensive book, appropriate for graduate courses, which the authors themselves recognize is too large and unwieldy for typical the undergraduate course. The material is covered very much in depth, and various compiler instructors have characterized using it in a course as the "read 10 pages, skip 30" approach. The second edition of this book was published in 1986, and given the decline of the compilers course, the authors themselves doubt there will ever be a third edition.
- Frustration with the Red Dragon book has led many authors to pen their own versions. However, it is safe to say that all other textbooks consist of that author's favorite subset of the topics in the Red Dragon book, *in the same sequence* [5,6,7,8,9,10,11,12,13,14,15,16,17, 18,19]. In the next section, we detail our own completely different sequence of topics.
- Regardless of textbook, the typical compilers instructor finds him/herself unable to cover all of the material in one semester, and with two-semester compiler course sequences being essentially extinct, the problem of too much material remains. To solve this problem, instructors resort to various schemes. They emphasize, de-emphasize, or flatly leave out topics in a haphazard way, often based on their familiarity (or unfamiliarity) with the topics. With fewer and fewer instructors being compiler experts themselves, the decisions being made are often not the best pedagogical ones.

The principal problem with compiler courses in the past has been precisely that – the focus on compilation. The underlying principles of translation, including syntax recognition and semantic processing, *transcend compilation.* In addition, the decreasing popularity of the compilers course is due, in our opinion, to the mismatch between the sequence of course topics, and the sequence of implementation efforts. We now proceed to describe our solution to this problem.

IV. The New Design

Here we present our new course design. It is based on the following premises:

- We rename our course design as Computer Translation with Applications. Although compilation is still one of the most common (and dominant) applications, we intend to remove a good number of compiler-specific issues, such as code generation for an actual processor, code optimization, register allocation, and floating-point arithmetic. We will also introduce the application of translation in other areas of Computer Science, such as command-line processors and user interfaces.
- The course design revolves around the implementation project. The project consists of maintaining and

extending an "initial" compiler, rather than implementing a compiler from scratch. The initial compiler is the implementation of an imperative C-like language that is *minimal*. The students extend and maintain the compiler, adding new constructs as the term progresses.

- In the new course design, we cover only the topics that are fundamental to translation, such as parsing, symbol management, and generation of code for a simple stack-based abstract machine.
- We specifically eliminate the traditional goal of having the students implement a "real" compiler.

Table 1 shows a side-by-side comparison of the old and new topic sequences.

	Old Design	New Design
1.	Introduction	Introduction
2.	Language description	Initial Language Description
3.	Lexical Analysis	Initial compiler Description
4.	Syntax Analysis	Translation of Operators
5.	Semantic Analysis	Translation of Statements
6.	Code Generation	Translation of Data Types
7.	Code Optimization	Translation of Subprograms
8.	Run-time Structures	Translation of Arrays
9.	Final Code Emission	Translation of Structures
10.		Applications

Table 1. Compilers/Translation Course Design

In the old approach, the topics are covered in the order in which the compiler itself proceeds with its work. This order happens to match the order in which an experienced compiler write might go about his/her implementation task, which (by the way) is one reason this order has rarely, if at all, been questioned in the past.

In the new approach, the students are given a complete working compiler for a minimal, imperative, C-style language. The language has variables of data type integer and boolean, and the ability to declare them. The language contains an assignment statement, a `while` statement, an `if` statement, and a `print` statement. The initial compiler implements only the unary minus operator, and the binary addition and ≤ operators. An intrinsic `read` function is used for input.

The implementation of the compiler will use `yacc` (or similar software for Java and C#), with a preprocessor program that translates regular right-part grammars into the pure context-free descriptions typically required by those software packages. Figure 1 shows the syntax of the initial language, which we call `Tiny`.

```
Tiny       -> 'program' Name ':' Dclns Block '.'
Dclns      -> 'var' (Dcln ';')+
           ->
Dcln       -> Name list ',' ':' Type
Type       -> 'integer'
           -> 'boolean'
Block      -> '{' Statement list ';' '}'
Statement  -> Name '=' Expression
           -> 'output' '(' Expression ')'
           -> 'if' '(' Expression ')' Statement
                              'else' Statement
           -> 'while' '(' Expression ')' Statement
           -> Block
Expression -> Term
           -> Term '<=' Term
Term       -> Term '+' Primary
           -> Term
Primary    -> '-' Primary
           -> 'read'
           -> Name
           -> '<integer>'
           -> '(' Expression ')'
Name       -> '<identifier>'
```

Figure 1. The syntax of Tiny.

The initial compiler will have a highly extensible and modifiable design. Implementations are under construction in C++, Java, and C#, to maximize the flexibility of the instructor.

The new approach resembles a spiral: students repeatedly visit every component of the compiler (scanner, parser, contextual constrainer, code generator), to add new constructs or features. With each visit, the student gains deeper understanding of the translator's architecture, components, and structure. Thus, we progress from the simple concepts to the complex, rather than from "front" to "back" of the compiler.

The differences between the two approaches are described in the Tables 2 and 3.

	Lexical Analysis	Syntax Analysis	Static Semantics	Code Generation
Operators	●			▶
Statements	●			▶
Data Types	●			▶
Functions	●			▶
Arrays	●			▶
Structures	●			▶

Table 2. Traditional Compiler Course Design

The disadvantage of the traditional approach is quite evident: each topic, say, syntax analysis, must be understood by the student in its entirety in order to implement the various constructs in the language. Furthermore, the implementation of the lexical analyzer is done all at one time, for the entire language, before moving on to the next compiler component.

	Lexical Analysis	Syntax Analysis	Static Semantics	Code Generation
Operators	●	●	●	●
Statements	↓	↓	↓	↓
Data Types	↓	↓	↓	↓
Functions	↓	↓	↓	↓
Arrays	↓	↓	↓	↓
Structures	▼	▼	▼	▼

Table 3. New Compiler Course Design

In contrast, in the new approach, the entire translation process of, say, operators, is discussed and implemented, before moving on to the next (more complex) language construct. This involves modifying the entire compiler, from front to back. Early on in the course, e.g. for operators, the student mimics what he/she sees already implemented in the compiler. Later in the process, when the student's understanding and mastery of the workings of the compiler have improved, the student will be ready to handle the more complex constructs. Perhaps the biggest advantage is that the student experiences a working, functional compiler from the first day, and the successful student project keeps it that way.

The architecture of the compiler writing system is shown in Figure 2.

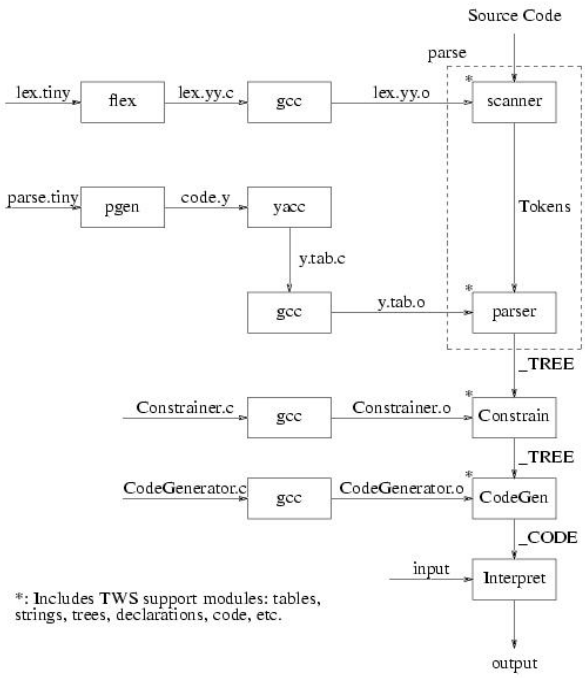

Figure 2. Architecture of the Translator Writing System

The current version of the system, written in C, utilizes `lex` and `yacc`. The grammar for Tiny, shown in Figure 1, allows regular expressions in the right-hand-sides of the production rules. This grammar is transformed into a pure context-free grammar, in the (arcane) notation suitable for yacc, by a preprocessor program called `pgen`. This program also generates, automatically, the C code necessary for construction of the parse tree.

V. Applications

Although compilation is the principal example of computer translation, there are many situations outside of compilers in which the central principles of translation are useful. Examples of this include data mining, command-line processors, translation of various markup languages such as XML, and both graphical and non-graphical user interfaces.

Part of the on-going research reported here involves seeking out prime examples of such applications, and incorporating their discussion into the design of the course.

Pedagogically, there is a potential additional windfall. It is well known that Computer Science graduates often leave the university without ever having implemented a truly large piece of software, say, with more than 20,000 lines of code. The reason is simple: no course or even course sequence can reasonably make such demands on a student's time. Still, a recurring theme among Computer Science educators is the question of where to obtain the *practicum*, i.e. how to expose the student to a well-written, well-structured, maintainable, good quality, large piece of software.

I believe a properly structured translators course can be a vehicle through which students acquire their first experience with a truly large program, by performing an extensive amount of maintenance on it, and by addressing the issues of redesign and software reuse in a large program.

VI. Conclusions

We have presented our new design of a computer translation course for Computer Science undergraduates. The new design is a radical departure from the traditional design. Rather than discussing the topics in the order in which the compiler does its job, we discuss the topics in increasing order of complexity. We also intend to keep only those topics that are fundamental to CS, such as syntax recognition, and discard most highly specialized, compiler-specific topics.

References

[1] ACM Computing Curricula, Final Draft, December 15, 2001, www.computer.org/education/cc2001/final

[2] M.E. Lesk and E Schmidt, Lex - Lexical Analyzer generator, http://dinosaur.compilertools.net

[3] Stephen C. Johnson, Yacc – Yet Another Compiler-Compiler, http://dinosaur.compilertools.net

[4] Aho, A. Sethi, R., and J. Ullman, Compilers – Principles, techniques and Tools, second edition, Addison-Wesley, Reading, Massachussetss, 1986.

[5] Steven John Metsker, Building Parsers With Java, Addison Wesley, 2001.
[6] Watt, D., and Deryck Brown, Programming Language Processors in Java, Prentice Hall, 2000.
[7] James Holmes, Building Your Own Compiler with C++, Prentice Hall, 1995.
[8] Fraser, C., and David Hanson, A Retargetable C compiler: Design and Implementation, Benjamin Cummings, Redwood City, California, 1995.
[9] Jim Holmes, Object-Oriented Compiler Construction, Prentice Hall, Englewood Cliffs, New Jersey, 1995.
[10] Pittman, T., and James Peters, The Art of Compiler Design: Theory and Practice, Prentice Hall, 1992.
[11] Fischer, C., and Richard Leblanc, Crafting a Compiler with C, Benjamin Cummings, Redwood City, California, 1991.
[12] Allen Holub, Compiler Design in C, Prentice Hall, Englewood Cliffs, New Jersey, 1990.
[13] Peter Rochenberg, A Compiler Generator for a Microcomputer, Prentice Hall, Englewood Cliffs, New Jersey, 1989.
[14] Peter Lee, Realistic Compiler Generation, MIT Press, Cambridge, Massachusetts, 1989.
[15] Fischer, C., and Richard Leblanc, Crafting a Compiler, Benjamin Cummings, Menlo Park, California, 1988.
[16] Peter C. Capon, Compiler Engineering Using Pascal, Macmillan, 1988.
[17] Tremblay, J., and Paul Sorenson, The theory and Practice of Compiler Writing, McGraw-Hill, New York, New York, 1985.
[18] Arthur Pyster, Compiler Design and Construction, PWS Publishers, Boston, Massachusetts, 1980.
[19] Barret, W., Bates, R., Gustafson, D., and John Couch, Compiler Construction, Theory and Practice, second edition, Science Research Associates, Chicago, Illinois, 1979.