

# Design and Implementation of a Modern Compiler Course \*

William M. Waite  
Assad Jarrahan  
Michele H. Jackson  
Amer Diwan  
University of Colorado  
Boulder, CO 80309  
USA

{William.Waite,Assad.Jarrahan,Michele.Jackson,Amer.Diwan}@Colorado.edu

## ABSTRACT

Current literature states that the undergraduate curriculum can no longer afford the luxury of a traditional compiler construction course. Nevertheless, there is an increasing need for an understanding of how to design and implement domain-specific languages. This paper presents a modern course in compiler construction, designed to provide a student with the capability of quickly constructing robust processors for a variety of language-related applications.

## 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, abstraction, tools, student culture, programming languages/paradigms

## 1. INTRODUCTION

The disparity between processor and memory speeds on current and future hardware platforms means that numerical algorithms must be designed to minimize memory traffic. A designer must predict the memory usage pattern, and then attempt to improve its properties by changes in the algorithm. Several days are required to do such a prediction

\*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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*ITICSE'06* June 26–28, 2006, Bologna, Italy.  
Copyright 2006 ACM 1-59593-055-8/06/0006 ...\$5.00.

by hand, limiting the amount of the design space that can be explored.

In the course of his research into the effects of memory usage patterns on performance [16], John Dennis built a source-to-source translator that accepted a MATLAB prototype of an algorithm and produced a version with additional code to predict that algorithm's memory usage. In this way, he was able to make accurate predictions in 20 minutes. The key point is that the source-to-source translator was a tool rather than the main object of his research.

This example is typical of a wide variety of situations in which a person needs to perform some kind of analysis or translation as an adjunct to their main project [32]. Very few (if any) of our students will write conventional compilers, but most will need to develop compiler-like processors to solve such ancillary problems. We believe that a modern compiler construction course should prepare students for these tasks [28].

In order to provide that preparation, the course must build an understanding of general principles and how those principles apply in a variety of scenarios [19]. Section 2 describes our goals in more detail, and explains how we selected the material to reach those goals.

Students come to a course with certain expectations and biases [30]. Section 3 explains the structure we used to convey the course material, and how it was based on these factors.

The philosophy and implementation described in this paper represents a significant modification of our previous compiler construction course, and was piloted in the fall semester of 2005 [2]. Section 4 presents our experience with that pilot.

## 2. COURSE MATERIAL

A consensus on the general principles of compiler design has developed over the last 45 years, and these principles are applicable to a wide variety of translation problems. They provide designers with a vocabulary to describe the essential aspects of a problem, and to concentrate on the decisions that are not purely mechanical. The principles are well-supported by abstractions, specification languages, and tools.

ACM's Curriculum 2001 [20] defines the course CS240s, Programming Language Translation, as follows:

Introduces the theory and practice of program-

ming language translation. Topics include compiler design, lexical analysis, parsing, symbol tables, declaration and storage management, code generation, and optimization techniques.

Lexical analysis, parsing, and symbol tables are implementation techniques that support general design principles. We believe that a modern compiler construction course should be at a more abstract level. Our goal was that students completing our course would be able to develop languages for specific problem domains, including design of appropriate syntax and relevant semantics. They would understand the decomposition of a language-processing problem into components whose characteristics could be described formally, be familiar with the corresponding notations, and be able to use tools to create robust processors for solving such problems.

Declaration and storage management, code generation, and optimization techniques imply a compiler whose target is a specific computer and must therefore assume a model of computation for that computer. That model embodies the instruction set, the procedure calling convention, the strategy for up-level addressing, and possibly garbage collection and exception handling. As we shall see in Section 4, a good understanding of computer architecture and assembly language is needed to grasp such a model.

Because that understanding is superfluous for the kinds of tasks described in Section 1, we chose to omit the treatment of specific models of computation and concentrate on the following design principles:

- Phrase and tree structure (concrete and abstract syntax)
- Computations over trees (relating structure to meaning)
  - Name analysis (scopes, defining and applied occurrences)
  - Type analysis (types, type equivalence, expression contexts)
- Output structure (context-free and context-sensitive patterns)

These principles abstract from the implementation techniques enumerated in the definition of CS240s. By casting the material in these terms, we move the students away from the familiar task of coding in some language and force them to think at a higher level.

The Eli compiler toolkit [3] can construct an economically viable translator from specifications of high-level decisions, thus obviating the need for coding in a programming language. The specifications we used to support our design-oriented approach are:

- Regular expressions and context-free grammars (phrase and tree structure)
- Attribute grammars (computations over trees)
- Aspect-oriented computational roles (name analysis, type analysis)
- Text patterns (context-free output structure)
- Tree grammars (context-sensitive output structure)

Our approach was to use these specifications in our explanations of the design principles, and to describe various possible design decisions by writing the corresponding specifications. Processors implementing those decisions could then be generated immediately and their behavior observed.

### 3. COURSE STRUCTURE

Students in a computer science curriculum belong to an occupational community [27] that creates and sustains a work culture involving task rituals, standards for behavior, and work practices. In order to teach a subject effectively, we need to understand the culture that characterizes the students' occupational community [31]: Classroom strategy, assignment selection, and assessment must reinforce the patterns that contribute to class goals and weaken those that do not.

#### 3.1 Classroom strategy

Our class of 21 students met for one hour each Monday, Wednesday, and Friday morning. The Monday and Friday meetings were in a normal classroom. Wednesdays we met in a lab that had 30 workstations; the students also had unlimited access to this lab outside of class.

We used each Monday classroom session to motivate a set of related decisions and illustrate various possible ways to specify these decisions. During the Wednesday lab session the students were given a realistic scenario and asked to write specifications to yield particular outcomes. They would then use Eli to generate a processor from those specifications, and verify its behavior. We reviewed this experience during the Friday classroom session, discussing common mistakes and extending the set of decisions and specifications.

Both the Monday and the Friday sessions were conducted as conversational classrooms [29]: The course outline listed specific topics to be covered and preparatory reading, but the responsibility to engage and participate in their own learning process was placed upon the students by fostering interaction and discussion rather than providing a lecture. Similarly, laboratories involved a rich interaction among students, professor, and teaching assistant.

Our use of two different kinds of rooms was important because it kept the two distinct kinds of activities we used separate in the students' minds. It also meant that the Monday and Friday discussion sessions had the students' complete attention, without the distractions of e-mail and web surfing, and that each room was configured appropriately for the activity being conducted.

Computer science students tend to boldly display their own opinions, and disqualify the opinions of peers [23]. This communication pattern leads to an environment characterized by competitiveness rather than cooperation, and condescension rather than understanding. It involves particular and recognizable types of discourse which, when prevalent in a classroom, can preclude the development of a collaborative and supportive learning environment [11]. We made every effort to avoid these types of discourse, encouraging open discussion and discouraging negative remarks.

#### 3.2 Assignment selection

Assignments determine the whole tenor of a course. Not only do they exercise the students on the course material, they reinforce more general goals and help students to avoid

common pitfalls. Assignment selection is therefore a vital component of the course design.

In this course, we wanted assignments to encourage collaboration. We wanted them to help the students to balance their workload over time, and to engender a more structured work process. Finally, we wanted them to be fun — to make the students passionate about the material.

### 3.2.1 Encouraging collaboration

Research has shown that students have an overwhelming preference for working alone [23]. The basic reason is that they regard assignments as products, for which they are paid with a grade [13].

We tried to counter this attitude by reducing the weight placed on assignments, and encouraged (but did not require) students to work together on all aspects of the course. By using the conversational classroom strategy for the Monday and Friday sessions, we validated collaboration as an important technique for improving understanding. In the lab, we tried to foster community by encouraging the students to help one another with problems. We pointed out that collaborative efforts arrive at better outcomes more rapidly [8], and spent an entire class period going through a structured group decision process [26].

### 3.2.2 Balancing workload over time

Procrastination is a common ritual of computer science students. It is often cast as a calculated risk, a game in which the student uses their superior skill to complete an assignment even though they have begun perilously close to the deadline [23].

To forestall this behavior, we made the first four assignments self-contained, straightforward applications of material discussed in class and practiced in the lab. It was therefore quite easy to predict the amount of effort required, and highly unlikely that a student would hit a snag that they had not seen before. This meant that there was no ego boost associated with pulling off a coup at the last minute.

The remaining assignment, which extended over half of the semester, was a project of the students' own choosing. The only constraints on this project were that it be a translator of some kind that required name analysis and had structured output. Projects ranged from conventional compilers to an aid for defining characters in Dungeons and Dragons (a role-playing game [1]).

There were three milestones, due at two-week intervals, in addition to the final submission: a proposal, an abstract syntax tree design, and a complete analyzer. These milestones limited procrastination, but more importantly they allowed for reflection about each step of the process. We provided detailed feedback, and were able to spot problems that individual groups were having with concepts and implementation.

### 3.2.3 Engendering structured processes

Students in programming courses approach assignments by “diving in” and “tinkering”, rather than attempting first to understand the problem [23]. This ritual is consistent with most assignments in such courses: There are many ways to obtain a particular result, and searching the solution space involves only things the students already know how to do. Getting a good understanding of the problem, however, may require them to learn additional concepts.

Because the students in our course are using specification languages to describe their understanding of a problem, rather than programming languages to describe how it is solved, there is no solution space to explore. Thus the student must come to grips with the problem itself in every assignment.

Most tasks assigned in programming courses can be carried out in complete ignorance of any process. Students therefore shun process as something laid down by the faculty rather than engendered by the task at hand. Thus they fall back on the experimentation ritual, believing that this flexibility allows them to complete the assignment in the most efficient way. Effectively, they are operating at the Initial stage of the Capability Maturity Model [14].

A comprehensive toolkit like Eli embodies significant knowledge of how to construct compilers, and therefore it forces a process on its users: Component problems must be understood in a certain order, because the understanding of one depends on decisions made when understanding others. Without the results of those decisions, subsequent problems cannot even be stated completely. Moreover, the vocabulary that must be used in describing the problems to be solved requires the students to think about those problems in certain ways. The overall effect, in conjunction with the assignments and milestones, is to raise the students to the Managed stage of the Capability Maturity Model.

### 3.2.4 Having fun

The compiler course has a history of cancellations due to low enrollment, and one semester all of the students attending withdrew before the drop deadline in October! We therefore made a significant effort to promote the Fall, 2005 pilot, and to capture and retain student interest by using a variety of scenarios for assignments and lab exercises. Our emphasis was on how to apply compiler technology to real-world problems with which the students could identify.

We also hoped a self-selected project would be more fun than a project designed by the instructor. This hope was realized, with students expressing excitement about their evolving processors and anxious to share their triumphs and show off their latest versions. Initial registration for the course was 28; eight dropped and one added to make the final count 21.

## 3.3 Assessment

Students were assessed on the basis of their assignments (30%), a midterm examination (30%), and their project (40%). The purpose of assessing assignments was to give the students motivation to carry them out and formal feedback on their approach to simple problems. They were required to demonstrate their individual grasp of the basic principles on the midterm examination.

Project assessment was arranged both to motivate the students and to give them an opportunity to reflect on and improve their performance: Each milestone was submitted electronically, and due at 9:00 on a Tuesday morning. It was evaluated by the professor and TA, with feedback given in the lab session the next day. Students were then allowed to submit an updated version, addressing any issues raised by the professor or TA, on Friday. The final version of each milestone was then graded and posted on the class web site.

## 4. RESULTS

The 21 students submitted 11 projects [2], only three of which were solo efforts; no project involved more than three students. Despite our omission of specific models of computation (Section 2), four of the projects were conventional compilers. Three of these used SPIM [22] as their target language and one used SRC [18].

The four conventional compiler projects were the easiest to assess: We already had conformance and deviance tests for the source languages (COOL [9], Tiger [10], SOOL [12], and Mystery [17]), so validation was simply a matter of running those tests. Although Eli supports the literate programming paradigm [21], only one of the groups made effective use of this facility by incorporating full documentation into their specification. One other group incorporated minimal documentation, and two provided separate documentation. One of those last interspersed their specification with their documentation by hand — a tedious and error prone process that would lead to maintenance headaches in the future.

Two of the four compilers were very successful, one partially successful, and one unsuccessful. Success clearly depended on recent experience with assembly language programming and/or computer architecture, which allowed the group to quickly develop an adequate model of computation on their target machine. Without that experience, the students found it extremely difficult to decide on the code to be produced. In the future, we would strongly urge a student not to attempt a conventional compiler unless they had a good understanding of the relationship between their source language and assembly code at the start of the project.

All but one of the remaining seven projects turned out well. Even the relatively unsuccessful group was able to come close to their stated goals, and we believe that their main problem was a lack of time on task.

Students greatly appreciated the freedom to choose their own projects within the guidelines mentioned in Section 3.2.2. It was apparent from discussions with several groups that this freedom did, indeed, allow them to do something about which they felt passionate.

Unfortunately, one group exploited the freedom of choice to turn their project into a normal programming assignment that did not demonstrate the principles we were attempting to teach. They proposed a source-to-source translator that would accept scanner and parser specifications for a language, along with a small file defining colors for highlighting [6] particular constructs of that language. Name analysis would be required because of relations among the specifications, and the highlighting definition files required by Vim [7] and Emacs [4] constituted the structured output. Thus this project satisfied the constraints stated in Section 3.2.2.

They also proposed writing a second version by hand in Python [5], and comparing the generation and hand-coding methodologies in detail. This was a seductive idea, since there is only one existing comparison of an Eli-generated program with a hand coded equivalent [25]; because the students involved were highly competent, we accepted their proposal with this addition.

Although they reached the second milestone successfully, it was clear that one team member was strongly resistant to specifying behavior instead of coding. They were unable to reach the third milestone (the complete analyzer gener-

ated from specifications), and argued that they would have nothing to show unless they were allowed to submit only the Python version.

On reflection, this was as much a failure on our part as on theirs. We did not sufficiently guard against the expectation that compiler construction would be regarded as a “programming” course just because it involves software implementation. Students have a number of expectations for a programming course, among them the idea that if they are good programmers they can get through the course on that skill alone. There is little need for them to be concerned with the specifics of the material, because those specifics are nothing more than what their programs must do. They believe that a programmer’s job is to write code to solve problems that they may not fully understand. We should have therefore been continually on the lookout for student strategies by which our goal of understanding might be compromised by a student’s goal of merely implementing.

## 5. CONCLUSION

We believe that compiler construction is an enabling technology for improving the computer’s ability to function as an intelligent assistant. There is a growing need for domain-specific languages to allow users to easily describe processes in terms of large operations and complex data [15]. Thus we favor the pedagogic strategy that presents the topic in terms of support for communicating with a computer [28].

Any course design must select the material to be included and (perhaps more importantly) the material to be omitted. Because the general principles involved in understanding a text in some source language and transforming that text into an arbitrary form are sufficiently complex that they require a full semester to master, we agree with the recommendation [24] to move the discussion of models of computation, code generation, and optimization to a following course.

The treatment of compiler construction should be kept at a high level by using formalisms to describe decisions and tools to implement them. This is not a typical programming class, and the instructor must be vigilant to prevent students from treating it as such. The tendency to implement rather than understand is only one of the aspects of student culture that a successful course design must deal with.

While assignments and laboratory exercises focus the students’ attention on specific, important aspects of the compiler construction problem, a project requires them to deal with all aspects great and small. They are faced with the consequences of their design choices, good or bad, and must decide when to go on and when to backtrack. We believe that this experience is vital.

Allowing the students a free choice of project within constraints, rather than having everyone do the same project, has both advantages and disadvantages. The obvious advantage is that the student is more passionate about a project that they have selected themselves. Some students may have difficulty thinking up a project, or may be reluctant to press for their own ideas. They may also fail to grasp the fact that something they are interested in can actually be cast in an acceptable form, as was the case for the Dungeons and Dragons project.

One obvious disadvantage of free choice is that a student may manage to manipulate the situation to circumvent the instructor’s goals for the course. Another is that some students will be unwilling or unable to come up with an idea for

a project. Both of these issues require additional effort on the part of the staff: More intensive monitoring of projects on the one hand, and time-consuming exploration of possibilities on the other. Nevertheless, our experience leads us to prefer free choice.

Finally, we suggest that a compiler construction course that uses the model discussed in this paper be limited to 25 students, and have at least one teaching assistant who is familiar with the tools and formalisms used. This student/staff ratio results in a reasonable response time in the laboratory, assuming that students collaborate and groups help each other with problems. It also enables us to carry out the necessary assessments of project milestones in one day so that we can give feedback in the laboratory.

## 6. REFERENCES

- [1] Dungeons and dragons. <http://www.wizards.com/default.asp?x=dnd/welcome>.
- [2] ECEN 4553. <http://ece.colorado.edu/~ecen4553>.
- [3] Eli: An integrated toolset for compiler construction. Documentation, examples, download from <http://eli-project.sourceforge.net/>.
- [4] GNU Emacs. <http://www.gnu.org/software/emacs/>.
- [5] Python. <http://www.python.org/>.
- [6] Syntax highlighting. [http://en.wikipedia.org/wiki/Syntax\\_highlighting](http://en.wikipedia.org/wiki/Syntax_highlighting).
- [7] Vim. <http://www.vim.org/>.
- [8] M. L. J. Abercrombie. *The Anatomy of Judgement*. Hutchinson, London, 1960.
- [9] A. Aiken. *Cool: The Classroom Object-Oriented Language*. <http://www.cs.berkeley.edu/~aiken/cool/>.
- [10] A. W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, Cambridge, UK, first edition, 1998.
- [11] L. J. Barker, K. Garvin-Doxas, and M. H. Jackson. Defensive climate in the computer science classroom. In *Proceedings of the 33rd ACM Technical Symposium on Computer Science Education*, pages 43–47, New York, 2002. ACM Press.
- [12] K. B. Bruce. *Foundations of Object-Oriented Languages*. MIT Press, Cambridge, MA, 2002.
- [13] G. Button and W. Sharrock. Project work: the organisation of collaborative design and development in software engineering. *Computer Supported Cooperative Work*, 5:369–386, 1996.
- [14] CMMI Product Team. CMMI for Software Engineering, Version 1.1, Staged Representation (CMMI-SW, V1.1, Staged). Technical Report CMU/SEI-2002-TR-029, Software Engineering Institute, Carnegie-Mellon University, 2002. <http://www.sei.cmu.edu/publications/documents/02reports/02tr029.html>.
- [15] J. Cohen. Updating computer science education. *Communications of the ACM*, 48(6):29–31, 2005.
- [16] J. M. Dennis. *Automated Memory Analysis: Improving the Design and Implementation of Iterative Algorithms*. PhD thesis, University of Colorado, Boulder, July 2005.
- [17] A. Diwan, W. M. Waite, and M. H. Jackson. PL-Detective: A system for teaching programming language concepts. In *Proceedings of the 35th ACM Technical Symposium on Computer Science Education*, pages 80–84, New York, 2004. ACM Press.
- [18] V. P. Heuring and H. F. Jordan. *Computer Systems Design and Architecture*. Pearson Education, Upper Saddle River, NJ, second edition, 2004.
- [19] 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.
- [20] Joint Task Force on Computing Curricula. Computing Curricula 2001 — Computer Science. Final report, Dec. 2001. [http://www.computer.org/portal/cms\\_docs\\_ieeeecs/ieeeecs/education/cc2001/cc2001.pdf](http://www.computer.org/portal/cms_docs_ieeeecs/ieeeecs/education/cc2001/cc2001.pdf).
- [21] D. E. Knuth. *Literate programming*. Center for the Study of Language and Information, Stanford, California, 1992.
- [22] J. Larus. SPIM: A MIPS32 simulator. <http://www.cs.wisc.edu/~larus/spim.html>.
- [23] P. M. Leonardi. The mythos of engineering culture: A study of communicative performances and interaction. Master's thesis, University of Colorado, Boulder, 2003. <http://www.cs.colorado.edu/~pltools/pubs/Leonardi.pdf>.
- [24] 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.
- [25] A. M. Sloane. An evaluation of an automatically generated compiler. *ACM Transactions on Programming Languages and Systems*, 17(5):691–703, Sept. 1995.
- [26] E. Triantaphyllou. *Multi-Criteria Decision Making Methods: A Comparative Study*. Kluwer Academic Publishers, Boston, 2000.
- [27] J. Van Maanen and S. R. Barley. Occupational communities: Culture and control in organizations. In B. M. Staw and L. L. Cummings, editors, *Research in Organizational Behavior*, volume 6, pages 287–365. JAI Press, 1984.
- [28] W. M. Waite. The compiler course in today's curriculum: Three strategies. In *Proceedings of the 37th ACM Technical Symposium on Computer Science Education*, New York, 2006. ACM Press.
- [29] W. M. Waite, M. H. Jackson, and A. Diwan. The conversational classroom. In *Proceedings of the 34th ACM Technical Symposium on Computer Science Education*, pages 127–131, New York, 2003. ACM Press.
- [30] 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.
- [31] K. Weick. *The social psychology of organizing*. Addison-Wesley, 1979.
- [32] D. E. Yessick and J. Jones. Reinventing the wheel or not yet another compiler compiler. In *Southeast ACM Conference*, 2002. <http://citeseer.ist.psu.edu/705273.html>.