Designing the Programming Assignment for a University Compiler Design Course

I. Budiselić*, D. Škvorc* and S. Srbljić*
* University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia
{ivan.budiselic, dejan.skvorc, sinisa.srbljic}@fer.hr

Abstract - This paper analyzes the key properties of a programming assignment for a university compiler design course. After an overview of popular choices used at several universities in the world, we describe the assignment given to students of the *Programming Language Translation* undergraduate course at the University of Zagreb, Faculty of Electrical Engineering and Computing, and provide the rationale behind several design decisions through an analysis of its evolution in recent years. While most compiler programming assignments are based on tools like lexer and parser generators, our students actually build slightly simplified versions of these tools themselves. We discuss our experiences with this programming assignment over the last three years and qualitative differences in the results between this assignment and the tool based assignment we had been using before.

I. INTRODUCTION

Translation of programming languages and compiler technology is one of the topics that are ubiquitous in computer science undergraduate curricula. It is usually implemented as a single course in the curriculum that covers some portion of the *Programming Languages* knowledge area (KA) and a few topics from other KAs in the *Curriculum Guidelines for Undergraduate Degree Programs in Computer Science* published jointly by the ACM and IEEE [1].

However, while most of these courses cover a similar set of topics in lecture, there is no broad consensus on the design of the programming assignment that is an integral part of almost every compiler design course. In fact, this has been the case ever since this topic was included in the curriculum [2].

The common characteristics of programming assignments in most of the compiler design courses analyzed in [1] are that students build a complete compiler, and the success rate on the project significantly impacts the students' grades. Apart from this general commonality, the organization and scope of the compiler project diverge significantly.

Compiler courses are typically thought in the second half of the undergraduate program with prerequisites covering programming, basic algorithms and data structures, computer architecture and organization and sometimes introduction to the theory of computation. Since the compiler bridges the gap between the two abstractions of high-level programming languages and machine code, this order of courses in the curriculum ensures that students are familiarized with both of these abstractions prior to taking the compiler design course. Furthermore, a large part of a compiler is based on well understood and approachable formal models, and tools for building compilers are abundant. These factors put the compilers course in a somewhat unique position in the curriculum in that students can build a complete and complex system in their programming assignment. Consequently, the compiler project is frequently the largest and most challenging programming project students are involved in during their studies. A few papers mention that students talk about their compiler project experiences in job interviews [3,4], and in our own experience, we've found that a student's performance on this project provides valuable information for writing recommendation letters.

It is the scope of the project that allows the high level of diversity that is observed in practice. Specifically, the design of a compiler project is based on five somewhat competing goals, and different preference for these goals yield significantly different designs.

First, students should ideally get some insight into all stages of the compiler, and the project should reinforce the theoretical material presented in lecture [3,5,6,7]. Understanding the compiler is beneficial to students as it is one of the most frequently used tools in the software developer's toolbox.

Second, compiler technology has applications beyond translating higher-level programming languages into machine code, and students should ideally experience that in their project so that they can recognize opportunities for applying the technology in other domains of their professional work [6,7,8].

Third, the complexity of the project provides an opportunity or even the requirement to teach students software engineering best practices in software design, testing, the use of tools such as compilers and debuggers, scheduling work and other areas [3,4,6].

The fourth goal that is often encountered is teaching students teamwork and communication as the project is nearly universally done in groups [3,4,9]. This seems particularly important as industry feedback frequently suggests that students lack key teamwork skills [9].

Finally, the fifth goal that is often not explicitly mentioned is that creating a compiler can make students better programmers. Knowing how some language feature

is implemented in a compiler allows students to better understand the programming languages they actually use. For example, if the source language supports passing arguments by value and by reference, implementing this distinction in the compiler requires and reinforces complete understanding of the underlying concept.

In this paper, we discuss the challenge of balancing these goals through our experience with the compiler project in the *Programming Language Translation* (*PLT*) course at the University of Zagreb, Faculty of Electrical Engineering and Computing (UniZg FER).

The rest of the paper is organized in the following way. In Section II, we overview some related work on compiler design courses. In Section III, we overview the compiler project at seven leading computer science universities in the US and Europe. We describe the curriculum context of our *PLT* course in Section IV. In Section V, we discuss the organization of the *PLT* course and describe the evolution of the programming assignment from its initial to its current design. We conclude the paper in Section VI.

II. Related Work

Two useful classifications of compiler design courses and projects have been discussed in the literature. Schwartzbach [10] roughly divides compiler design courses into *front end heavy* and *back end heavy*. The former category focuses on formal languages and context free parsing, while the latter focuses on code generation, resource management and optimization for a realistic processor architecture. Semantic analysis and intermediate code generation, which are typically considered a part of the front end [11], are grouped into the *middle end* of the compiler, and both course categories cover these phases to some extent.

Under this categorization, a meaningful critique of both types of courses is that lexing and parsing are supported by easy-to-use tools, and that the compiler back end is typically reused as a component when building a compiler for a new language. Specifically, the bulk of the work in creating a new compiler is done in this middle end, i.e. in semantic analysis and intermediate code generation. Schwartzbach suggests that both front end heavy and back end heavy courses are somewhat constrained to using only simple languages in compiler projects. Instead, the paper presents a *middle end heavy* course and argues for projects that are based on large, feature-rich languages.

Waite [6] proposes three strategies for organizing the compiler design course and project that are mostly in line with the goals presented in the introduction. The *software project* strategy focuses on teaching software development techniques rather than compiler algorithms. To achieve this goal, compiler building tools are employed as much as possible. It is unclear which parts of the compiler should be thought in lecture and at what level of detail, but this strategy seems to stray away from a traditional compilers course in this sense.

Quite oppositely, the *application of theory* strategy focuses on theoretical models such as formal languages and attribute grammars. The goal of a course following this strategy is to teach students both the value and limitations of theory in computer science. The suggested programming assignments in this strategy are *tools based on theory*, which seems to imply that students should only build parts of a compiler.

Finally, the *support for communicating with a computer* strategy focuses on the applicability of compiler technology to many domains in software engineering. After completing such a course, students would be able to devise domain specific languages for various applications and use the compilers toolkit to implement these languages. Again, this strategy does not include a compiler project. Instead, students practice language design and use of tools.

III. Overview of the Compiler Project at Leading Computer Science Universities

In this section, we briefly overview the compiler design courses and accompanying programming assignments from some of the leading computer science universities in the US and Europe. The specifics of each course are shown in Table I.

We've estimated the percentage of lecture time devoted to lexical and syntax analysis (shown in the third column in Table I) from the courses' syllabuses available on their web sites. The three courses with no more than 15% of lecture devoted to these topics are either middle end or back end heavy by Schwartzbach's classification. They provide only a practical introduction to lexing and parsing with the goal of enabling students to either write or modify provided specifications for existing lexer and parser generators that are used mostly like black boxes. The remaining four courses with at least a quarter of lecture time devoted to the lexer and parser additionally cover the generator algorithms themselves at the expense of more advanced optimization approaches in the back end, but should not be classified as front end heavy.

All of the courses except the Columbia University course use fixed source and target languages in the compiler project. All of these source languages are object-oriented and typically syntactically and semantically resemble the language that is used in the introductory programming course in the curriculum. Similarly, the target architectures are typically those used in other parts of the curriculum as well.

Fixing the implementation language provides several potential benefits. First, it is easier to estimate the amount of code students will need to write to complete the assignment. Second, having a fixed implementation language makes it easier to provide starter code which can then be used to adjust the difficulty of the assignment without sacrificing the overall functionality of the compiler. Third, a specific lexer and parser generator can be used for that fixed implementation language and potentially covered in lecture or recitation. In fact, all seven courses use existing generators for these early phases of the compiler.

The project is usually done in small groups or even individually. Columbia uses the largest groups of five

TABLE I. KEY PROPERTIES OF COMPILER COURSES IN SOME OF THE WORLD'S LEADING COMPUTER SCIENCE UNIVERSITIES

University	*Course name*	*Lecture front end %**	*Source language(s)*	*Target language(s)/ architecture(s)*	*Implementation language(s)*	*Tools*	*Students per group*	*Project grade impact %*
MIT	6.035 Computer Language Engineering	10	Decaf	x86-64	Java	ANTLR	3-4	60
Stanford	CS 143 Compilers	30	COOL	MIPS	C++, Java	Lexer and parser generators	1-2	50
CMU	15-411 Compiler Design	10	L1-L4	x86-64	SML, Ocaml, Haskell, Java and others	Lexer and parser generators	1-2	70
Berkeley	CS 164 Programming Languages and Compilers	30	COOL	MIPS	Java	JLex, CUP	1-2	40
Columbia	COMS W4115 Programming Languages and Translators	25	Student designed	Student choice	OCaml	ocamllex, ocamlyacc	5	40
Oxford	Compilers	15	Oberon-like	Keiko/ARM	OCaml	ocamllex, ocamlyacc	N/A	N/A
ETH Zurich	Compiler Design I	30	JavaLi	x86 or similar	Java	JLex, CUP	2	66

* Approximate values

students with an emphasis on group organization [3]. On the other hand, groups of two students are most common, and students are often encouraged to solve the assignments with pair programming instead of splitting up the work.

Finally, the project accounts for between 40 and 70 percent of the students' grade in the class. To grade the project, most courses use a combination of objective testing where individual assignments are tested for correctness against a set of test inputs and either oral quizzes or extensive written documentation.

IV. THE CURRICULUM CONTEXT OF *PLT* AT UNIZG FER

Before discussing the design of the compiler project in our *PLT* course at UniZg FER, we will describe the position of the course in the computer science curriculum as it is an important factor in choosing the goals of the course and the design of the compiler project.

The *Programming Language Translation* course is taught in the fifth semester of the undergraduate curriculum. Students enrolled in the course previously complete an introductory programming course and a course on basic algorithms and data structures, a computer architecture course and a course covering the introduction to the theory of computation.

Both the introductory programming course and the algorithms and data structures course are taught using C and a small subset of C++ (basically C with classes). In the computer architecture course, students learn FRISC and are introduced to ARM. FRISC is a RISC processor architecture developed at Unizg FER specifically for teaching processor architecture. The *Introduction to Theoretical Computer Science* (*ITCS*) course extensively covers regular and context-free languages that are of particular interest for compiler construction.

Besides these enabling factors, there is one characteristic of the curriculum which makes the design of the compiler project more challenging. While students have an opportunity to learn several other programming languages through electives and skills, there is no single language that every *PLT* student is certainly familiar with except C. The costs and benefits of C make it a poor choice for the implementation language in a compiler course.

V. *PLT* COURSE ORGANIZATION AND GOALS

About a third of lecture time in *PLT* is devoted to lexical and syntax analysis. This provides students with an opportunity to apply what they've learned about automata theory in the previous *ITCS* course to two interesting problems which illustrates the general value of these models. Furthermore, understanding the algorithms behind lexer and parser generators makes it easier to avoid ambiguities when designing a language or to eliminate them if they arise.

The second third of the course covers semantic analysis and intermediate representations. Based on their programming experience, students generally have an intuitive understanding of the key topics in semantic analysis, such as declarations and type checking. By introducing attributes to the familiar model of a grammar, this intuitive understanding can be formalized in an approachable way. Similarly, attributed syntax trees can be easily understood through their relationship with parse trees that are also covered in *ITCS* and in the first part of *PLT*.

The final third of the course is dedicated to code generation and execution support, along with a limited introduction to both machine independent and machine dependent optimization. In this part, students draw from their experience writing FRISC and ARM programs.

The primary goal of the compiler project is to reinforce these concepts introduced in lecture with practical experience. In the remainder of this section, we describe the evolution of the compiler project in the last

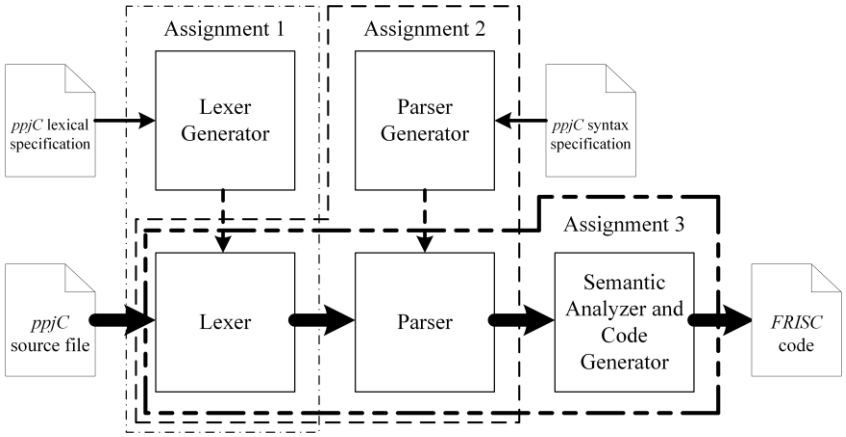

Figure 1. The design of the first iteration in the evolution of the compiler project in the *PLT* class at UniZg FER.

four years, from its initial organization to the one we are currently using.

A. *Lessons Learned from an Ineffective Compiler Project Design*

Up until 2010, the compiler project we used was fairly free-form, as students would design or choose their source, target, and implementation languages. This project design is intended to provide students with an opportunity for language design and familiarization with the compiler building tool chain, i.e. the second goal mentioned in the introduction is emphasized.

To prevent procrastination, we would set up two check points during the semester—one for lexical and syntax analysis and one for semantic analysis. At these check points student groups would demonstrate their current progress and group members would be quizzed on what they had done. At the end of the semester, groups would make their final code submission along with written documentation describing their source and target language, compiler architecture and key implementation ideas. Each group was also required to prepare a set of examples that showcase the capabilities of their compiler.

A teaching assistant would then read the documentation, run the provided examples, examine the compiler's source code, and come up with additional test cases. As a grading aid, we used an extensive list of possible language features, each of which was assigned some weighing factor corresponding to the difficulty of implementing it. This list included the most basic features like lexical analysis and support for variables and integral data to fairly advanced features like classes, inheritance and optimization.

After this preparation, the groups would come in for the final lab quiz focused on code generation, and the teaching assistant assigned to each group would go over the compiler's feature set with the group leader to make sure something wasn't overlooked.

The final scores were then scaled so that the most successful group would get 7.5% of the grade per member. The group leader was given the total group score and had to assign up to 7.5% to members, ideally depending only on their contribution to the project's success. The three lab quizzes contributed 2.5% of the grade each, so the project contributed a total of 15% to the grade.

While this project design allowed perhaps a few top students per generation to enjoy designing and implementing their own language, over several years of using it we noticed many significant problems.

The key problem was that at least half the groups didn't manage to finish the project. Difficulties usually arose during semantic analysis and code generation. This is understandable as lexical and syntax analysis were supported by *Lex* and *Yacc* equivalents, and appropriate definition files for many languages are available online. To get these stages working, students often simply modified existing generator input. This in turn made the start of the assignment very easy, giving students a false sense of the overall difficulty of the project and the time required to complete it.

Second, students were typically unsure about many design decisions in the early assignments as they didn't have a clear picture of what exactly they would be doing in later assignments. The most problematic point in this sense was the symbol table design which students often attempted to introduce in the lexer only to find later that their design was too limited.

Third, while students enjoyed the freedom to design their own source language, they often didn't choose an appropriate feature set. In particular, students often underestimated the complexity of some feature they wanted to include. Designing "the best" and most complex language became a goal for some groups, which often caused problems in completing the project.

Fourth, as the grading process was time consuming for teaching assistants and between 150 and 200 students take this class every year, we were forced to keep groups fairly large, usually between 10 and 12 students per group. Managing such a large group was understandably difficult for the group leaders, and this resulted in significantly different experiences for various members of the group. Usually, only the leader and a few other members would work on the compiler code, while others were assigned to creating test cases, writing the documentation or other supporting roles.

Finally, students often complained that the project is far too difficult for only 15% of the total grade. Furthermore, while we made every effort to keep project grading fair, this was a challenge as different teaching assistants graded different projects with different source

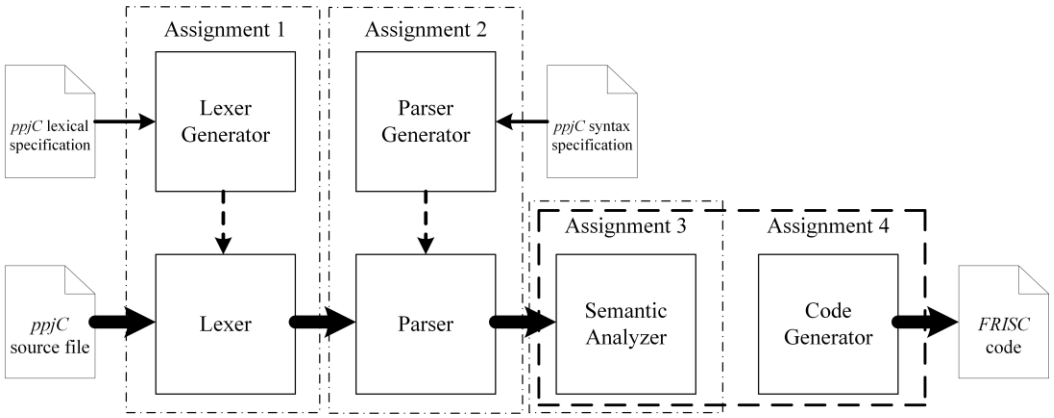

Figure 2. The design of the current compiler project in the *PLT* class at UniZg FER.

and target languages in an inherently subjective process. Specifically, it was difficult to correctly assign partial credit for features that weren't completely implemented, which was a common occurrence.

B. *Evolution of the Compiler Project - the First Iteration*

With the downsides of the old project in mind, in 2010 we set out to design a new project with the following ideas. First, we attributed most of the issues in the project to *underspecification*—students were given too many choices that they objectively weren't equipped to make. Most notably, we decided to specify both the source and the target language. We also included extensive advice on some key implementation details in the assignment document which was ultimately 60 pages long.

Second, we reduced the group size to nine students. While we were aware that the groups were still too large for effective organization, we had decided to make changes incrementally to avoid unexpected problems. For example, there was a concern that reducing the group size too much would result in some groups not having any strong programmers to lead the effort.

The design of this first iteration of the project is shown in Fig. 1. Unlike most other courses, we decided to have students implement their own lexer and parser generators instead of using existing ones. There are several motivating factors for this choice. First, all the algorithms required to write these generators are covered in lecture. Implementing these algorithms reinforces this knowledge. Second, the early assignments are made more challenging which is beneficial in that it doesn't portray a false picture of the complexity of the project as a whole, and students usually have more time to devote to projects early in the semester. Finally, building your own version of a tool forces you to understand it better, and hopefully teaches more than simply copying and editing existing lexical or syntax specifications.

With this in mind, the first assignment was to build a lexer generator that was a slightly simplified version of *Lex*. We provided students with lexical specifications for several example languages and for *ppjC* which was the source language we designed. We chose C as the basis for the language as that is the language that all of our students are exposed to early in the curriculum. The feature set of *ppjC* included a simplified integral and floating point type system, loops, branches, functions, single pointers, one dimensional arrays and structs. We intentionally kept the language fairly simple so that a majority of the groups would be able to complete the project.

The second assignment was to create a parser generator. We chose to use the *LR(1)*-parser as it allows the simplest and most intuitive grammars for specifying language syntax. Again, we simplified the syntax specification compared to a tool like *Yacc* so that the input is basically just the language grammar. As in the first assignment, we provided the syntax specification for several languages including *ppjC*. The students had to integrate the generated *ppjC* parser with the lexer they had generated in the first assignment.

Finally, the third assignment was to implement semantic analysis over the generated parse tree and generate *FRISC* code. We chose *FRISC* as the target processor as it is familiar to our students, which allows them to focus on the compiler itself. The semantics of *ppjC* were defined using natural language, mostly relying on the students' knowledge of C. Similar to the second assignment, students were again expected to integrate the new code with the results of the previous assignments.

As part of the third assignment instructions, we gave students a list of language features with weight factors, much like the one we had been using to grade the old project. The students were required to submit a similar list with the features they had implemented and test cases that exercise those features along with their documentation.

Overall, the changes we made felt like an improvement. While some students complained about the size of the assignment instruction text, nearly all groups managed to develop high quality solutions to the first two assignments. However, it was also clear that the third assignment needed significant additional improvement as many groups still stumbled in either semantic analysis or code generation. We identified two main reasons for this. First, compared to the first assignments, the third assignment was both underspecified and too big. Second, the fact that we required integration of all stages at every step made it difficult to recover from errors in earlier stages. For example, a small error in the parser generator made it nearly impossible to successfully finish the project.

C. The Current Design of the Compiler Project

In the next offering of the course, we made an effort to address these remaining problems. We split the third assignment into two separate assignments as shown in Fig. 2. Additionally, we completely removed the dependencies between all the assignments except the third and fourth. This means that the ability to complete an assignment is not negatively impacted by errors in earlier assignments. We achieved this by making the output of each assignment be the exact input for the next assignment. Specifically, the lexer outputs the token sequence of the input *ppjC* program which the parser transforms into a parse tree. The parse tree is then the input to both the semantic analyzer that outputs semantic errors and the code generator which is implemented as an extension to the semantic analyzer.

To further assist students in semantic analysis, we specified most of the semantics of *ppjC* in a formal way, using an attributed translation grammar, i.e. by adding attributes and computation rules to the *ppjC* grammar. The attribute values computation rules are designed for recursive descent on the parse tree, which is also covered in lecture. With this extension and additional implementation advice for the code generator, the assignment specification document grew to 80 pages, but we observed a significant improvement in the success rate of these later stages of the compiler which was our primary goal.

To address the grading problems, we introduced automatic evaluation of all assignments. Students submit their solutions to the web-based evaluator, which then runs several integration tests to check if the solution archive is properly formatted and correctly handles input and output. After the assignment deadline, all solutions are evaluated using 20-30 focused test cases that attempt to exercise various features of the particular stage of the compiler. The results of these tests contribute up to 15% of the student's grade. With the additional maximum 5% for each of the two assignment-oriented oral quizzes, we've increased the total grade impact of the project to 25%.

To reduce group management problems, we reduced the group size to six students. Additionally, to enable automatic evaluation, we constrained the implementation language choice to C, C++, C#, Python and Java. These are the languages the students are most likely to encounter in the curriculum depending on their course choices. Nearly all groups choose either Java or Python, with several groups choosing C# and C++ each year.

We also simplified *ppjC* by removing floating point numbers, pointers and structs. While this made the type system significantly simpler, it still offers a good insight into type checking and conversions.

With this design, we've achieved our primary goal of making sure most groups manage to implement at least some semantic analysis and code generation. In the last three years, the evaluation success rate was above 70% for the third assignment and above 50% for the fourth assignment, and only several groups fail to score any points on these assignments. The success rate for the first two assignments is consistently over 70%.

VI. Conlusion and Future Directions

In this paper, we've described the evolution and current design of the compiler project in our university compiler course. With a project with detailed specification of all the basic phases of the compiler, we've achieved significantly higher success rates in the later stages of the compiler than with the more free-form project we had been using in the past.

We've been getting positive feedback from students regularly, but many students still find the assignment too difficult. We are considering addressing this problem with designing two distinct assignment paths, where one would be a lower-credit version of the assignment focusing only on some parts of the compiler and language design. Additionally, we are also planning to further reduce group size which should minimize management problems, and more importantly, make the project experience similar for all students.

Acknowledgment

We would like to thank Ivan Žužak and Zvonimir Pavlić for their significant contribution to the improvement of the course and the compiler project in the last several years.

References

[1] The Joint Task Force on Computing Curricula, "Curriculum guidlines for undegraduate degree programs in computer science," ACM and the IEEE Computer Society, 2013.

[2] B. Appelbe, "Teaching compiler development," Proceedings of the 10th SIGCSE technical symposium on computer science education, vol. 11(1), pp. 23–27, 1979.

[3] A. V. Aho, "Teaching the compilers course," ACM SIGCSE Bulletin, vol. 40(4), pp. 6–8, 2008.

[4] W. G. Griswold, "Teaching software engineering in a compiler project course," Journal on educational resources in computing (JERIC), vol. 2(4), 2002.

[5] J. S. Mallozzi, "Thoughts on and tools for teaching compiler design," Journal of computer sciences in colleges, vol. 21(2), pp. 177–184, 2005.

[6] W. M. Waite, "The compiler course in today's curriculum: three strategies," Proceedings of the 37th SIGCSE technical symposium on computer science education, vol. 38(1), pp. 87–91, 2006.

[7] M. Mernik and V. Zumer, "An educational tool for teaching compiler construction," IEEE Transactions on Education, vol. 46(1), pp. 61–68, 2003.

[8] M. Ruckert, "Teaching compiler construction and language design: making the case for unusal compiler projects with PostScript as the target language," Proceedings of the 38th SIGCSE technical symposium on computer science education, vol. 39(1), pp. 435–439, 2007.

[9] W. M. Waite, M. H. Jackson, A. Diwan, and P. M. Leonardi, "Student culture vs group work in computer science," Proceedings of the 35th SIGCSE technical symposium on computer science education, vol. 36(1), pp. 12–16, 2004.

[10] M. I. Schwartzbach, "Design choices in a compiler course or how to make undergraduates love formal notation," Compiler construction: 17th international conference ETAPS 2008, pp. 1–15, 2008.

[11] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, "Compilers: principles, techniques, and tools,", Pearson Education, 2006.