Developing an Optimizing Compiler for the Game Boy as a Software Engineering Project

Stefan Kögel, Michael Stegmaier, Raffaela Groner,
Stefan Götz, Sascha Rechenberger, and Matthias Tichy
Institute of Software Engineering and Programming Languages, Ulm University
D-89069 Ulm
<firstname>.<lastname>@uni-ulm.de

ABSTRACT

Software engineering students not only have to learn theoretical concepts but also how to successfully apply them in practice. Hence, projects are an important part of software engineering curricula. As software engineering methods and technologies are only relevant for non-trivial software systems, we report in this paper on a master-level software engineering project course in which a team of students developed a compiler for the Game Boy in a single semester. The students developed different languages and corresponding parsers to a common intermediate language, optimizations on the intermediate language, as well as a code generator for the Game Boy. We particularly present lessons learned by us and the students as well as potential course improvements.

CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**; • **Software and its engineering** → *Compilers*;

KEYWORDS

Compiler construction, software engineering education, project-based learning

ACM Reference Format:
Stefan Kögel, Michael Stegmaier, Raffaela Groner, Stefan Götz, Sascha Rechenberger, and Matthias Tichy. 2018. Developing an Optimizing Compiler for the Game Boy as a Software Engineering Project. In *ICSE-SEET'18: 40th International Conference on Software Engineering: Software Engineering Education and Training Track, May 27-June 3 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183377.3183388

1 INTRODUCTION

Getting experience in realizing reasonable-sized software engineering projects should be an important part of software engineering education. This stimulates not only learning of fundamental software engineering knowledge and skills as well as their application in realistic settings, but also experience in team work.

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE-SEET'18, May 27-June 3 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5660-2/18/05…$15.00
https://doi.org/10.1145/3183377.3183388

Experience with using projects in engineering education, i.e., project-based learning [5], shows that students achieve a better understanding of how to apply knowledge, gain problem-solving skills, and show higher motivation. This in turn assists their learning.

Complementary to software engineering skills, those projects also give students the opportunity to gain technical knowledge in specific areas.

In this paper, we report on a course which applies project-based learning to the area of compiler construction, a topic which is often not in the core of software engineering curricula. However, it is on the one hand well understood in theory and practice and on the other hand compiler construction knowledge can be useful in various areas, from parsing complex network messages to the development of domain specific languages. Our particular focus in the project was to teach modern software engineering principles like working in small teams, continuous integration, and giving presentations.

Our course uses the Game Boy as a target device for the compiler. The Game Boy is (as seen from today) a rather simple device. Hence, a group of students can create a compiler from scratch for it in a reasonable time frame. Furthermore, our students still relate to the device from their own childhood which sparks interests and increases motivation.

During the course, the group of students develop (1) own programming languages and parsers to an intermediate language, (2) complementary optimizations on the intermediate language, and (3) a code generator to the Game Boy. Figure 1 contains an overview of the modules developed by students.

In the next section, we present the course design with more details including learning objectives as well as the course structure. Section 3 discusses our lessons learned and planned improvements. After a short presentation of related work in Section 4, we conclude and sketch our future plans in Section 5.

2 COMPILER CONSTRUCTION PROJECT

2.1 Learning Objectives

Our learning objectives were twofold: students should get experience in applying software engineering principles and gain an understanding of compiler techniques.

Knowledge and understanding: Students should be able to explain the advantages of using version control systems, integration tests, and clearly defined interfaces between modules. Furthermore, students should be able to explain how the different steps of a compiler pipeline produce optimized machine instructions. That is, what is a parser, what are common optimizations and

their trade-offs, and how can high-level language constructs be translated into low level machine instructions.

Skills and abilities: Students should be able to self-organize in small teams using version control software and perform quality assurance through integration tests and benchmarks. They also need to be able to present their work to other students. At last, they have to be able to implement a parser using a technology of their own choice, implement a common code optimization, and to generate assembler code for certain programming language constructs.

Judgment and approach: Small teams of students have to organize themselves and distribute complex tasks between their members. Students and teams also have to choose appropriate architectures for various tasks involved in writing a compiler.

2.2 Course Structure

Our course is structured into the three following phases. These are also shown in Figure 1:

Parser: Each student should define their own syntax and develop a parser for it.

Optimizers: The students should work out optimizations either alone or in small teams and implement them.

Code Generator: Two teams of students implement a code generator and a register allocation algorithm. The teams have to standardize on an interface between their implementations.

We structure the phases further by giving one lecture and one assignment per week. This gives students a goal to work towards from week to week.

For the **Parser**, we allow students to use any parsing technology of their choosing and make no restrictions on the syntax of their language. Our only requirement is that the students' programming languages are able to express: arithmetic expressions, assignments, arrays, loops, if-statements, and procedure definitions.

The students' parsers have to serialize their parsed abstract syntax tree into an XML format defined by us (see Section 2.3 for details.) This enables a modular design that allows any parsers to be used with all optimizers and code generators. The result of this modular architecture is that the work of all students can be combined into an optimizing compiler that supports all programming languages defined by the students.

To implement more complex **Optimizers**, students have to work in teams. They use a version control system (i.e. GitLab) with automatic unit and integration tests that ensure that all modules work together. Some of these tests are given, others are created by students, giving them experience in writing their own unit and integration tests. Students also have to benchmark the effects of their optimizers on the code generated by the compiler.

For developing the **Code Generator**, students are divided into two teams: One team creates a code generator that generates assembler code that uses an unlimited amount of, so called, magic registers. The other team implements a register allocation algorithm that calculates an efficient allocation of Game Boy hardware registers on the the magic registers. For this step students have to develop a low level intermediate representation that can represent Game Boy assembler instructions and magic registers.

Our compiler is made up of many modules that have to be run in a certain order. At first, we used several Python scripts to sequence these models in order to run our integration tests. Two students implemented a simpler to use and more robust *Pipeline Tool* that can be controlled via configuration files.

To keep each other informed about their used technology and progress, students have to give presentations about their work at the end of each phase.

2.3 Used Resources

In order to allow collaboration of students, we defined an intermediate representation in the form of an *Abstract Syntax Tree (AST)* as an XML Schema Definition. The parsers of the students have to output XML files complying with this XML Schema Definition.

To interface with the Game Boy hardware we have defined a Hardware Abstraction Layer in the form of magic functions that can be called the same way as regular procedures. These magic functions allow things like handling graphics, producing sound, reading inputs and so on.

We created an interpreter for the intermediate representation that enables the students to run their AST files. This allows them to test their parsers and optimizations without a working code generator. In the interpreter, we implemented the magic functions to behave the same way as they do on the Game Boy hardware.

To facilitate testing, we also provide magic functions for printing to the console. For this we provide a tool that emulates the CPU of the Game Boy and prints to the console when these magic functions are called. This way the students can compare the output produced by their generated code with the output produced by the interpreter. We set up git repositories with continuous integration for every student. With these git repositories the students can benefit from automated testing.

In the end, every student will write their own game in their own language that can be run in an emulator such as Emulicious [1] and on a Game Boy using a re-programmable cartridge.

The Interpreter, XML Schema, and description of the magic functions are available at https://drive.google.com/open?id=0B02lNuyg_dFLRGFNR2VVa3NjM28

3 EXPERIENCE REPORT

While it is essential for any kind of project to have a solid plan to go on, it is also necessary to adapt to shortcomings and improve the initial plan through experience.

Which is why in this section we take a look at the experiences we made during all phases of the project and analyze in which way we can improve the project for future generations of students. We will do so by evaluating our experiences of the strengths and flaws of our plan. First by looking at each phase individually and afterwards by inspecting the project as a whole.

3.1 Phase One: Parser

We planned to have students write an arithmetic expression parser as the first assignment, which turned out to be too much for a first assignment. Parsing binary operators with precedence rules is notoriously difficult. Furthermore, students had to become familiar with a parsing technology and our AST format. This did put us one

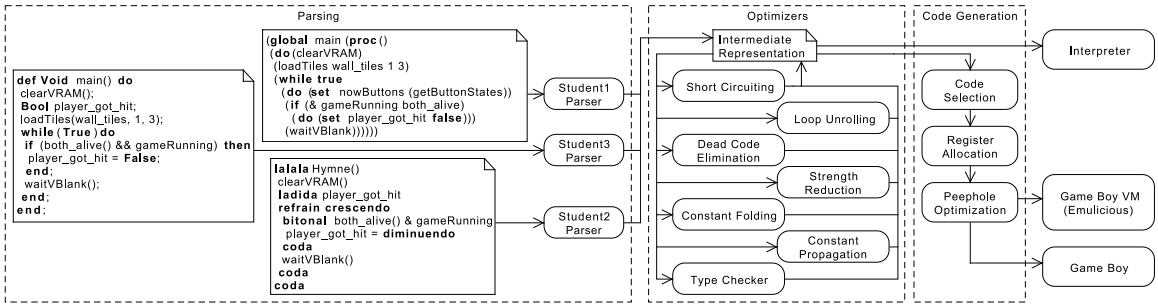

Figure 1: Phases and work packages per phase

week behind schedule. We therefore conclude, that the opening assignment should be more simple.

In the current version of the course, students have to parse number literals and procedure calls. This is much simpler and allows students to use the magic functions of the interpreter to draw simple pictures.

At the beginning of the project the used programming languages were: Java (6 students), JavaScript (1 student), C++ (1 student), Haskell (1 student), Elixir (1 student), and Prolog (1 student). Five students ended up using Java and three students Haskell for writing their parsers. The language choice did also transfer to the other two phases of the project which resulted in the Code-Generator being written in Java while the Register-Allocation was done in Haskell. We were also able to observe a pattern linking the choice of programming language with the choice of parser technology. All parsers written in Java were generated by using ANTLR[1] and all parsers written in Haskell were created by using Parsec[2].

This alignment of used technologies can be explained by students helping each other with their preferred technology and by having to use a common technology when working in a team.

Two students tried to write their parser completely from scratch. One of them switched from C++ to Haskell with Parsec and the other one dropped out. Thus, we strongly advice students against writing their parsers by hand.

Examples of the created languages can be seen on the left side of Figure 1. Most languages were Java-like with some changes to comments or to how different data types are called. Other students took inspiration from other languages (i.e. Haskell and LISP) or changed up how keywords were called completely. We attribute the Java resemblance of most created languages to both the students familiarity with languages of this style and the fact that our AST suggests an imperative programming language.

3.2 Phase Two: Optimizations

In order to parse our intermediate representation, we suggested the usage of already existing libraries for parsing XML data structures. Two students did not follow our suggestion and ended up having a hard time finding bugs in their own XML parser implementation. I.e. they did not handle XML comments or did not parse self closing tags correctly.

[1]http://www.antlr.org/
[2]https://wiki.haskell.org/Parsec

Another problem was that many students did not implement all features for their optimization that we requested. Students also invested too little time into benchmarking their optimizations. Both problems can be avoided by better communication on our side. For example, we plan to request integration tests for the optimizations before students implement them. In this way, we can check that they have understood all requirements of their optimization.

We also recognized that a way for students to check for side effect free code sections would have been helpful, because many optimizations can only be applied to side effect free code. Many students had to implement this by themselves with varying levels of success which further impacted the quality of the resulting optimization. Offering a tool that checks for side effect free code sections can improve the results of the final optimizations, while simultaneously simplifying their implementation.

During this phase two students also implemented a pipeline tool for sending an input program through a parser, all optimizations, the code generator, and a Game Boy assembler. All steps performed by this tool could be controlled via configuration files. The pipeline tool simplified the implementation of unit and integration tests, which were essential for finding bugs during the remaining project. We recognize how valuable this can be for quality assurance and will therefore reuse this setup in future projects.

Shortly after students started working on their optimizations for the second phase a voluntary lecture evaluation was performed by our university. Some students voiced their dissatisfaction with the fact that they were still not able to write code that could be run on an actual Game Boy. As a take away for our current project, we decided to switch phases two and three to have executable Game Boy code at about half way through the semester. We also hope that this switch could make it easier for students to realize the importance of optimizations when seeing the limits of what non optimized programs can achieve on hardware with resources as limited as those of a Game Boy.

Furthermore, the evaluation showed that students found that the project had a good structure, that the contents were reasonably easy to follow, and that all students would recommend the project to other students.

3.3 Phase Three: Code Generation

Our students were split into two groups for the last phase of the project. The first group implemented a Code generator for Game

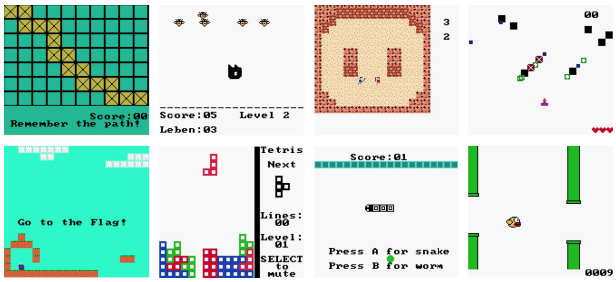

Figure 2: Screenshots of games created by our students. Screenshots taken from the Game Boy emulator Emulicious [1]

Boy assembly using an unlimited number of magic registers. The generated code would then be passed to the program developed by the second group of students which replaced access to magic registers with actual registers using a graph-coloring algorithm.

The main problem with this approach is that no program can be properly compiled for the Game Boy until the register allocation works correctly. Thus, in our current project, we are treating the Game Boy as a stack machine. This allows our students to generate valid Game Boy assembler without magic registers from the beginning. Register allocation can then be implemented as an optimization on the stack machine code.

Our plan of giving students complete control over how to organize within their respective groups ended up backfiring and causing discord, forcing intervention by us. While this can be an important learning experience about the importance of proper project organization and planning, it also causes unnecessary stress and friction between students. This can however be fixed by enforcing some stricter organization from the beginning.

3.4 Looking back

Throughout the whole project, we forced students to write unit tests for each completed feature of their parser, optimizer, and the code generator/register allocator. Our goal was to show students the importance of testing. However, the tests students created were not created with any kind of coverage in mind making the testing environment ineffective. This forced us to detach one advanced student for filling the gaps in test coverage. For future projects, we will have to enforce stricter organization and structure for tests. We will also take advantage of the already existing testing environment, by using it as a way of tracking the progress of our students.

Over the course of the project we also fell behind our planned schedule by four weeks, because we extended deadlines to allow students to catch up. While this practice provided a pleasant work environment, it failed in promoting efficient planning and deadline handling.

To prevent students from falling behind, we introduced a mandatory meeting each week in our current project. During this *Lab*, students will be able to work on the project while we are present to help with problems.

In the end, even though there were some problems along the way, eight students were able to use their self-developed languages to write games that could be run on a Game Boy COLOR. These games can be seen in Figure 2 and a short video of all finished games in action is available at https://youtu.be/kr_aJmJLJOc

4 RELATED WORK

There are several projects which discuss compiler construction but don't consider e.g. language design, optimizations or software engineering. Aiken [2], Daley [3], and Mernik and Zumer [4] present projects to teach students compiler construction by implementing a compiler as individuals or in teams. Those projects consider primarily teaching compiler construction without focusing on other aspects like, for example, software engineering or the need for students to work together in self-organized teams. Schocken et al. [6] present a course consisting of a set of projects which leads to building a computer. Students gain experience in software engineering, digital architectures, compilers and operating systems. In contrast to our project, students don't design their own language and optimizations aren't considered.

5 CONCLUSION

In this paper, we present a project that allows students to gain practical software engineering experience by implementing a compiler. During this project, students reported that they found this project very interesting and we felt that they had a lot of fun. This was despite the project requiring many hours of work per week from the students. We attribute this to the fact that students had a lot of freedom and could use languages and tools that interested them the most, in order to implement their compiler.

The project can be improved further by simplifying the first assignments, so that students can focus more on familiarizing themselves with parsing and continuous integration tools. Further, we propose *Lab*s to get better feedback about the progress and problems of our students. These *Lab*s also allow us to intervene when students get stuck on a problem.

REFERENCES

[1] 2017. Emulicious (Game Boy [..] Emulator). (2017). Retrieved October 4, 2017 from http://emulicious.net (Author: Michael Stegmaier).
[2] Alexander Aiken. 1996. Cool: A portable project for teaching compiler construction. *ACM Sigplan Notices* 31, 7 (1996), 19–24.
[3] James S Daley. 1978. A laboratory approach to teaching compiler writing. In *ACM SIGCSE Bulletin*, Vol. 10. ACM, 19–21.
[4] Marjan Mernik and Viljem Zumer. 2003. An educational tool for teaching compiler construction. *IEEE Transactions on Education* 46, 1 (2003), 61–68.
[5] Julie E Mills, David F Treagust, and others. 2003. Engineering education—Is problem-based or project-based learning the answer. *Australasian journal of engineering education* 3, 2 (2003), 2–16.
[6] Shimon Schocken, Noam Nisan, and Michal Armoni. 2009. A synthesis course in hardware architecture, compilers, and software engineering. In *ACM SIGCSE Bulletin*, Vol. 41. ACM, 443–447.