Practice of Compiler Course Close to the Needs of the Industry

How We Did and What We Have Learned

Yu Zhang HuanQi Cao Cheng Li
University of Science and Technology of China

ABSTRACT

With the rapid development of programming languages and modern computer systems, there exists a gap between the students' skills that can be trained by the practice of the compiler courses and the needs of the Industry. To close the gap, we designed a new practice framework and integrated it into the complier course organization, which takes advantage of new techniques such as widely adopted open source ANTLR and LLVM compilers, etc. The analysis of students' performance and feedback indicates that our design is effective and the complexity level of the practice projects is moderate. We also report lessons learned which will guide the future improvement.

KEYWORDS

compiler practice, LLVM, ANTLR

ACM Reference format:
Yu Zhang HuanQi Cao Cheng Li. 2018. Practice of Compiler Course Close to the Needs of the Industry . In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the recent decade, a large number of new programming languages have emerged and played an important role in developing various software and hardware [8], while the attention has been shifted from CISC and RISC to new architectures used by GPU, FPGA, etc. With the fast development of software and hardware, applications and systems in use by industry become extremely larger and more complex [2]. To adapt to these rapid changes, practitioners need a basic knowledge of compilers [1], as the compiler principles and technologies touches upon all these fields. However, as pointed out by Radermacher *et al.* [6, 7], there exists a gap between what students could learn from universities and the skills demanded by the Industry.

Faced with increasing and evolving programming languages as well as diverse target machines, it is worth rethinking the teaching objectives and contents of compiler course to meet the needs of industry. Compiler course should be designed to develop the following capabilities as the objectives: 1) to reason and explain the cause of compilation problems or phenomena in actual programming; 2) to design a language (especially domain-specific language or specification language) and rapidly implement it using existing tools; 3) to formally describe a language's lexical rules, syntax, even semantics; 4) to understand the types and architecture features of modern mainstream compilers; 5) to learn design patterns, teamwork, programming methodology, and software engineering tools by compiler projects. We repeatedly emphasize in the class that students need to form a habit of consulting the language specification and compiler manual to understand a programming language, rather than simply looking at its tutorial. All this expects a good compiler course design so that students are getting more chances to transfer the theoretical knowledge into practices.

To improve the compiler course design, we have to ask a very important question beforehand, which is whether the mainstream compiler course organizations adopted by universities are sufficient to achieve the above five objectives. To answer this question, we investigated the status quo of domestic compiler courses by conducting a questionnaire survey in 21 universities in Anhui Province. Through a comprehensive analysis, we found that unfortunately the practice of the compiler courses hadn't received much attention. For example, the hours spent by students in finishing practical projects are too short and the project design is too simple to make students gain a deep understanding of compiler principles and apply those knowledge in real world problems.

To address this limitation and close the gap between university training and industry need, in this paper, we primarily focus on reconstructing and strengthening the practice of the compiler course. To this end, we have carefully designed a practical compiler framework, which consists of several pluggable components and can be integrated with modern open source compilers such as LLVM [3]. Based on the framework, we have designed and carried out the following three kinds of course practice projects:

1. *Step-by-step compiler component experiments*, which form a compiler for a small language integrated with backend existed or developed by students. We recommend using ANTLR (http://www.antlr.org/) to construct the parser since it is widely used in academia and industry, and easy to add error handling into its top-down analysis (see Section 3.2.2 for more details).

2. *Reading comprehension tasks*, which require students to read selected source code of modern compilers (*e.g.* LLVM) guided by some tips provided, and then answer the given questions. Through these tasks, students are able to understand the architecture and features of modern compilers in limited time.

3. *Extensive projects*, whose topics are open and provided by the teacher or students themselves. The extensive project is a comprehensive project, which may be done in group. Through it, students can learn new trends in programming language area such as new language features and their mechanisms, or do some exploratory practical issues, etc.

Corresponding author: yuzhang@ustc.edu.cn.

Conference'17, Washington, DC, USA
© 2018 ACM. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

In Fall 2017, we integrated this practice framework and the corresponding projects into our compiler course, which had been enrolled by 42 bachelor students. In order to make students be aware of importance of compiler practice, we increase the proportion of experiments in the total score, up to 50%. Their performance and feedback indicate the effectiveness of the framework and project design, as well as its moderate complexity level. We hope our attempt can be seen as an option for other universities where instructors would like to change their compiler course organization. Different universities can choose part or all of the above kinds of projects according to their own circumstances. Through the well-designed practical framework and a series of projects, students would be cultivated to gradually get the above five aspects of capacities.

The reminder of the paper is structured as follows: Section 2 motivates the need to change the compiler practice by presenting the history and status quo of the compiler course. Section 3 presents the overview of the practical framework. Section 4 presents kinds of projects based on the framework. Section 5 reports the students' performance and feedback, summarizes the lessons learned and possible improvement policies.

2 BACKGROUND AND MOTIVATION

In this section, we first introduce the status quo of domestic compiler courses, including the questionnaire survey in Anhui province; then introduce some changes related to programming environment in the new situation.

2.1 Status Quo

The compiler course is a classic core course in Computer Science. It introduces how to design and implement compilers. The principles and techniques for compiler design are also applicable to so many other domains that they are likely to be reused many times in the career of a computer scientist [1].

However, with the mass higher education since 1998, many universities and colleges canceled the compiler course or reduced the class hours, and some of them did not set the course practice. In our university, since Grade 2000 the class hours of the compiler course reduced from 80h to 60h, and the independent compiler practice course of 40h was canceled. Furthermore, since Grade 2009 *CS top-notch innovative student training pilot program* has been carried out, and then an honer compiler course is added to enhance the depth and breadth of the course for some excellent students. The honer course has the same class hours as the regular one. Under the circumstance, we have designed different teaching content for two levels of compiler courses, the main difference between them is that we strengthen the course practice of the honor course.

In order to know the status of the compiler course in other universities, we designed a questionnaire consisting of a series of questions for the purpose of gathering information about the compiler courses offered by different universities in Jul. 2017. Till Aug. 2017, we collected 23 questionnaires from 21 different universities in Anhui. We performed a statistical analysis on these questionnaires by summarizing the number of students taking the compiler course, the hours for theoretical and practical teaching, as well as the type of experiments which questionnaire universities have chosen. Table 1-3 show the corresponding results, respectively.

Table 1: Statistics on the Number of Students in the Course

# of Students	0-50	51-100	101-150	151-200	251-300	351-400
N_q	1	10	5	4	1	2

N_q: number of questionnaires

Table 2: Statistics of Theoretical and Practical Hours

Hours' Range	0	6-12	16-20	32-40	42-48	51-54	64	Total
N_q (Theortical)	0	0	0	8	7	7	1	23
N_q (Practical)	4	6	9	4	0	0	0	23

N_q: number of questionnaires

According to the questionnaires, there are about 3360 students in 21 universities who take the compiler course each year. Table 1 lists the number of questionnaires, denoted as N_q, in the given range of student numbers, while the student number ranges where $N_q == 0$ are not shown. From the table, we find that the number of students taking the compiler course varies across universities.

The average theoretical and practical teaching hours in questionnaire universities are 45.3 and 15.9, respectively. As illustrated by Table 2, it is worth mentioning that 4 out of the 23 questionnaires didn't include any experiments. This result highlights that the practice in the compiler course hasn't received enough attention. As compiler experiments are extremely important for students to digest and understand the abstract and difficult compiler principles, with no or limited practice hours, it is very challenging to guide students to transfer their theoretical knowledge to address real world problems.

Table 3: Types of Experiments Chosen

Type	# Applying this Type
Lexer	16
Parser	16
Semantic Checking	3
Code Generation	3
Static Analysis	1
Implement Concrete Algo.	3
Simulate Concrete Algo. by Hands	3
Extensive Projects	1

We also look into the amount of theoretical content covered by the experimental setup used by the questionnaire universities. As depicted in Table 3, most of universities choose simple experiments which only cover a small fraction of theoretical content. For example, most of the questionnaire universities only introduce lexing and parsing into practice. Some of them even just let students to implement algorithms written in the textbook or simulate them by hands. Obviously, the incompleteness of the experiment design likely loses chances to comprehensively evaluate students' performance and cannot enhance their software engineering and problem solving skills.

2.2 Various Changes in the New Situation

In the past decade, programming language, hardware and software engineering and the development environment have undergone enormous changes. We summarize changes related to modern software development into the following categories.

1. *Programming languages flourish.* There are thousands of programming languages, and the popularity of these languages is constantly changing. For instance, the top three most popular languages in TIOBE Index for Jan. 2018 are Java, C and C++, but those in 1970s

are Fortran, Lisp and Cobol. In addition, old-timer languages such as C, C++ are constantly evolving. Their latest standards are C11 (ISO/IEC 9899:2011) and C++2014 (ISO/IEC 14882:2014).

2. *A wide range of target platforms.* Except the CISC and RISC architectures of the past, there are vector, VLIW, multicore, and various accelerators such as GPU, FPGA, and some novel storage devices such as NVM and RDMA. Modern programming frameworks like deep learning systems expect programming models that are highly productive and flexible w.r.t a wide range of machines.

The challenge introduced by the emergence and evolution of languages and target platforms is that it is impossible to cover in a compiler course the diversity of algorithms and techniques used in modern commercial compiler.

3. *Open Source Software Prevails.* With the rapid growth of open source software, more and more people and organizations use public code, document and Q&A warehouses in the Internet to interact and collaborate with each other, *e.g.* through GitHub, GitBook and Stack Overflow. The GitHub Octoverse 2017 indicates that 24 million people and 1.5 million organizations across 200 countries work across 67 million repositories on GitHub. With so much code on GitHub, it is a natural place for people to learn and prepare for careers in technology. Currently millions of teachers and students now work together on GitHub, but few come from China.

3 RETHINKING THE COMPILER PRACTICE

In order to adapt to the new situation mentioned in section 2, we need to rethink the objectives as well as the practical contents of the compiler course, and explain them in details in this section.

3.1 Capability Objectives

The **main capability objectives** of the course are as follows.

G1. Programming skills. Students need to understand underlying mechanisms and possible impact factors in compilers and target systems. Thus, they can quickly analyze and solve practical programming problems, as well as explain the phenomena.

G2. Formalization skills. Students need to understand principles of regular expressions and finite automata, context-free grammar and parsing techniques, even type system. Thus, they can formally describe the syntax and semantics of a language.

G3. Language design and implementation skills. Based on G2, students need further to understand syntax-directed translation, tree-based methods, code generation strategy, and storage organization, etc. They also need to be familiar with some compiler generator to design and implement DSLs or specification languages.

G4. Engineering skills. Through well-designed compiler practice system, students need to be able to work with software at large scale, consult literature, take advantage of tools, and develop soft skills about progress management, personal and team work, communication and document writing.

G5. Innovative mind. In the teaching process, through some well-designed homework and teamwork, students are encouraged to investigate and explore modern compilation and computer systems, then attempt to do some exploratory research.

Additionally, it is important to supervise students to form a habit of checking language specifications and compiler manuals to explain phenomena and problems in actual programming, so that students can cope with the evolution of languages and diverse implementations.

To achieve the above objectives, we think that the practice of the compiler course needs to be strengthened and elaborately designed.

3.2 Practice Framework

To build compilation experiments that can meet the needs of students of different levels in different universities, we design a compiler practice framework as shown in Fig. 1. We next present the design principles, open source software and tools selection, as well as the kinds of experiments in turn.

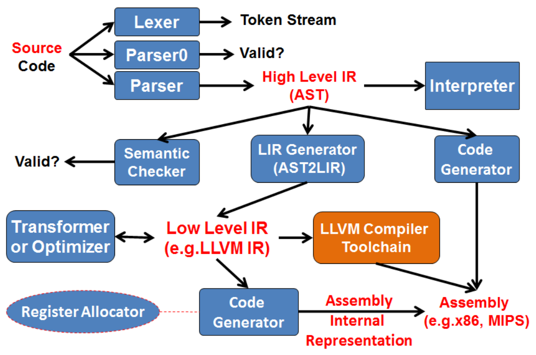

Figure 1: Overview of the compiler practice framework.

3.2.1 *Design Principles.*

The design of the framework follows two principles as below:

Step-by-Step Practice. To solve the issues of "hard-to-start" and "no or small practices", all course experiments should be designed by following the "step-by-step" principle. On the one hand, this can make the experiments be synchronized with the theoretical teaching and help students to understand theoretical knowledge deeply. On the other hand, through several step-by-step experiments, students can finally complete a substantial project close to the actual project size at the end of the course.

Decomposability and Pluggability. Since different universities have different teaching objectives, resources and requirements, we need to design a unified practice framework and develop some concrete systems containing some essential and optional components. These components should be pluggable, and a concrete system based on the framework could be decomposed, then reassembled into a new concrete system. The "decomposability and pluggability" principle is also critical to construct each university's curriculum practice system. This is because each university has the requirement for some adjustments to the curriculum system year-after-year to reduce plagiarism or to accommodate new circumstance.

3.2.2 *Open Source Software and Tools Selection.*

Here we focus on the choice of open source compiler infrastructure and parser generator, as well as software engineering tools.

Open Source Compiler Infrastructure. GCC and LLVM are the two most mainstream open source compiler infrastructures. The young LLVM began as a research project at UIUC [3], with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. It has grown to be an umbrella project consisting of a collection of modular and reusable compiler and toolchain technologies. Since LLVM is better than GCC in terms of documentation and modularization, and has been widely used in industry and academia[*], we choose LLVM and introduce it into the compiler class. Thus, a student can touch modern large-scale compilers and attempts to integrate LLVM with his own compiler.

Parser Generator. A parser generator is a programming tool that creates a parser from some form of formal description of a language and machine. Some parser generators are based on bottom-up parsing, *e.g.* YACC and its variants such as Bison and Java CUP; and some are based on top-down parsing, *e.g.* JavaCC and ANTLR. The advantage of bottom-up parsing (*LR-style*) is speed and expressiveness, while the advantage of top-down parsing (*LL-style*) is easy to understand and easy to add error handling. For a long time in compiler teaching, Lex and YACC are used to construct the parser. However, it is not easy to write grammars to fit the constraints of deterministic *LALR(k)*, especially when adding error handling. ANTLR is a powerful top-down parser generator which allows powerful, nondeterministic parsing by continuously exploring new theories and techniques, *e.g. LL(*)* [4] from ANTLR v3 and *ALL(*)* [5] from ANTLR v4. It is widely used in academia and industry to build all sorts of languages, tools, and frameworks. In view of this, we select ANTLR for compiler practice since 2017.

Software Engineering Methods and Tools. To meet the needs of the industry, we introduce several kinds of software engineering methods and tools in the course practice. For example, 1) *Version control system.* We create a separate git repository for each student since 2012 to develop their process and version management skills. 2) *Design pattern.* We have adopted design pattern such as visitor, singleton and factory to construct experimental software support library for more than a decade. We also try to make students understand these patterns via practice. 3) *Build tool.* We require students to use build tools such as make, cmake for a long time. 4) *Internet collaborative tools.* We create GitHub and GitBook repositories for the course since 2015, and guide students to use them and gradually work together using these platforms.

3.2.3 ***Various Kinds of Compiler Experiments.***

We have designed three kinds of course practice projects which can be applied in different levels of students and universities.

Table 4: Various Compiler Experiments

No.	Description	Level	ReadExp
S1	Start-up	Basic	
S2	Lexer	Basic	R1
S3	Parser0	Basic	
S4	Parser with AST generation	Basic, Opt.	R2
S5	Parser with Checker	Basic, Opt.	R3
S6	Interpreter	Basic, Opt.	
S7	LIR Generator	Basic, Opt.	
S8	Simple Code Generator	Basic, Opt.	
S9	Code Generator	Adv.	R4
S10	Transformer or Optimizer	Adv.	R5

1. *Step-by-step compiler component experiments* (S1 to S10 in Table 4), which finally form a compiler for a small language. Experiments S1 to S8 are basic, where S4 to S8 are optional to meet the needs of different universities' experimental choice. By completing the first three and several of the next five students would be able to obtain the basic knowledge and skills required for developing an interpreter, a parser or a compiler of an experimental language. S9 and S10 are advanced, which aim to challenge students to use knowledge and skills obtained from the class and extracurricular investigations for program analysis, optimization or code generator.

2. *Reading comprehension tasks* (R1 to R5 in Table 4, each of which refers to the topic listed in the corresponding of column 2), which require students to read selected source code of modern compilers (*e.g.* LLVM) guided by some tips, and then answer the given questions. Through these tasks, students attempt to understand the architecture and features of modern compilers in limited time. These experiments require students to read the relative source code of the actual compiler, then answer the given questions. The key to design these experiments is how to provide an effective code reading guide and well-designed questions. So students can start the reading experiment easily and spend limited time on the key points of the actual compiler.

3. *Extensive projects*, whose topics are open and provided by the teacher or students themselves. The extensive project is comprehensive and may be done in team. Through this kind of projects, students may learn new changes and trends in programming language area or attempt to solve some exploratory practical issues.

Table 5: Features of the language and their difficulty levels

Features	Level
Only integer data type, no function call, assignment, if, while	Basic
break, continue, more data types, functions w/o parameters	Medium or High
OO features	High

It should point out that different language features have different difficulty levels of implementation, since they require different depth of knowledge and skills. Table 5 lists some language features and their difficulty levels in implementation. There are more factors which need be further considered in the construction of a concrete practical compiler system, for example: (1) the style of the experimental language, which could be PL/0, COOL, a subset of C-like or Java-like etc.; (2) programming language for experiments, which could be C, C++ or Java; and (3) the form of the used IRs and relative APIs. We have developed and accumulated a lot of experimental support libraries for more than ten years.

4 EXPERIMENTS DESIGN

Following the guideline listed in Table 4, we design a series of experiments for complementing the theoretical teaching of the compiler course by taking into account the objectives of having course practice, students' skills and the time arrangement. Due to space limit, we collapse multiple step by step experiments into two major experiments. The first experiment covers the stages S1 to S4, R1 and R2, and relies on ANTLR. The second experiment covers S5, S7 and relies on LLVM. In addition, we prepare tutorials covering R3 to R5. Due to time limit, the remaining experiments are ignored.

[*]You can find LLVM users in http://llvm.org/Users.html, and related publications in http://llvm.org/pubs/

4.1 Implementing S1: Start-up

To help students quickly start their work, in this experiment, we offer a set of technical documents and a few tutorials to guide the students to get familiar with the development environment and a simple language to be translated. With regard to the environment, students are required to do some exercises to install and configure a variety of software such as Unix-style operating system, ANTLR v4 jar package, JDK and LLVM. In addition, we ask them to play with development utilities such as CMake for building example code, Git for organizing their code, and web-based repositories like GitHub and GitBook for material sharing. With regard to the simple language, we design the one that supports C1 – a very small subset of C, as all students learned C before.

4.2 Implementing S2-4 with R1&2: ANTLR v4

The primary goal of this experiment is to let students write their own parser which takes a piece of code written in C1 and generates the corresponding abstract syntax tree (AST). To help students achieve this goal, we offer an incomplete code structure in which we only implemented the AST definition and data structure while leave the rest undefined. With the support of ANTLR v4, students are supposed to first fill in the code structure the lexer grammar (a.k.a. S2) for guiding ANTLR to parse the input code into a series of lexical tokens. Second, they have to write the parser grammar (a.k.a S3) for guiding ANTLR to generate the respective parse tree from the tokens. Finally, they need to program a tree visitor (a.k.a S4) for traversing the generated parse tree and translate it into AST.

As this experiment intensively uses ANTLR, we assign students two additional reading tasks about ANTLR which corresponds to R1 and R2 in Table 4. With regard to R1, we make the students gain a basic understanding of ANTLR v4 by walking through a simple example, which parses an arithmetic expression with ANTLR grammar and calculates the value by an ANTLR Listener written in Python. Concerning R2, we suggest the students to carefully read the code implementing the AST data structure for C1 and the symbol table organization.

To evaluate the students' solutions, we also design a serializer which serializes an AST object into a Json object and outputs the serialization to a file. Then, we compare the file generated by students' code against the one by the reference code provided by us.

4.3 Implementing S5&S7: LLVM

The goal of the second major project is to let a student implement his own IR generator, which takes a piece of code written in C1 as input and translates it into the corresponding intermediate representation (IR). This project will reuse the parser implemented by the previous project to produce AST. To help students accomplish this goal, similar to the previous project, we offer students an incomplete code framework integrated with LLVM IRBuilder APIs. The remaining task for students is to fill in the missing AST visitor, which traverses the AST and build the corresponding LLVM IR module.

As this project relies on LLVM, before students start implementing their solutions, we offer students an opportunity to exercise the LLVM IR and the corresponding APIs through a warm-up phase, which consists of the following steps: (a) students learn about the format of LLVM IR and manually translate some simple C programs into LLVM IR; and (b) students study LLVM IRBuilder APIs and call these APIs to automatically generate the semantic-equivalent IR for a few more complex programs written in C++.

Compared to the first project, this one is more challenging and time consuming, and thus it requires us to give more guidance to students. To this end, we further divide this project into a few fine-grained steps and motivate students to make progress step by step. The order of steps is so important and unchangeable as the successor steps may depend on the predecessor steps. First, they need to construct a basic AST visitor which only generates functions with empty body. Second, we ask students to fill in functions to generate code for block statements and function calls, but leaving methods related to syntax structures of other types blank. Third, they need to think about the handling of expressions and assign statement. At the fourth step, students can write code to support variable declaration and type checking. Due to time limit, we only introduce two basic type checks to validate if a variable is a `CONST` or `ARRAY`. However, we leave space for students to complete semantic checking. The final step is to add code generation for control syntax such as `if-else` and `while`. One of the major benefits of having such step-by-step design is to enable students to test their IR code generator after finishing every step.

To evaluate the students' solutions, we additionally take advantage of a simple JIT execution engine offered by LLVM–MCJIT so that we can execute the generated LLVM IR module and compare the output to the one produced by our reference implementation.

4.4 Implementing R3-5: Various Tutorials

Beside the above coding projects and their associated reading tasks, we also provide students with two tutorials. As mentioned in Section 3, the primary goal of setting up these tutorials is to make students read source code of large-scale modern compiler systems and explore their new features. To fulfill this goal, with regard the first tutorial(a.k.a. R3), we assign students a task of reading Clang Static Analyzer and answering a set of given questions. For the second tutorial(a.k.a. a combination of R4 and R5), we guide students to build a simple Just-in-time compiling engine with layered structure based on LLVM OrcJIT predefined layers. This structure enables them to consider further optimization on the JIT process. It is worth mentioning that we do not require students to implement the related backend algorithms which are taught theoretically. Instead, they just need to get the big picture of the backend.

5 COURSE EXPERIENCE

In Fall 2017, to demonstrate the claims made before, we replaced the old project used by the Compiler course in previous years with the one presented in the Section 4. In this section, we report the statistics of students' performance, summarize the lessons we learned and propose future work for improving the compiler course.

5.1 Overall performance

In total, there were 42 bachelor students who had enrolled in the honer compiler course. We evaluated the outcome of students' labs by checking four aspects: (a) if their code was compliable and executable; (b) the correctness of functional features they implemented; (c) the goodness of the documentation with a focus on analyzing the

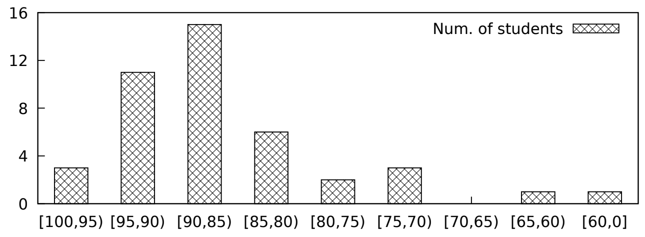

Figure 2: Number of students whose scores falling into each score range

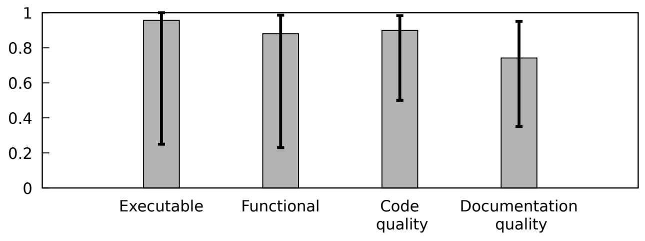

Figure 3: Normalized average score concerning each metric plus min and max values.

problems each lab needs to address and sketching their solutions; and (d) the code quality. Figure 2 summarizes the overall performance of these students and the score distribution. The biggest concern in our mind when designing the practice framework was that students may not be able to adapt to the new changes. However, the fact that majority of students achieve a score higher than 80 implies that the lab design is not so challenging that many students could easily bootstrap their work. The second concern we had was that the design might be too simple to differentiate the degree to which they mastered and applied knowledge learned from class. This doubt was eliminated by the fact that there exist a few students did not do well in their lab solutions and achieved a score under 80. As a result, we believe that the moderate complexity level of the practice framework is good classifying students' performance.

5.2 Performance breakdown

Next, we shifted our attention to investigate the students' breakdown performance on each metric basis. As depicted in Figure 3, the average score per metric is high, but the minimal score is very low. This result is consistent with the overall performance as most of students would be able to complete a large fraction of required work, while the remaining ones couldn't. Among the four metrics, students paid more attention to making code works rather than writing a coherent technical document. Concerning the importance of documentation, we would like to add a training lesson next year to teach students the principles of making plans before technically address problems and summarizing experience upon the completion.

5.3 Reported problems

For each lab, we additionally require students to write a document which consists of a basic description about their achievements as well as problems they identified during implementing their solutions. Here, we summarize the problems they reported so that we can understand well if the lab design is effective and learn lessons to improve it. Table 6 shows the types of problems and the corresponding short descriptions. Most problems are concerning the adoption of ANTLR, LLVM, and C++ features to implement the intended functionalities. This is mostly because students didn't get used to read a significant amount of materials. To address this, we plan to improve the lab documentation by shortening the content and highlighting the important points that we want students not to miss. Surprisingly, there were a quite large number of students (some of them hadn't submitted their problems explicitly but asked during office hours) faced challenges introduced by configuring working environments and utilizing tools, which have been intensively used by both academia and industry. With regard to this point, we plan to submit a proposal to the course management department which will encourage the instructors of the prerequisite courses of Compiler to put an emphasis on the practice of those environments and tools.

Table 6: Representative problems raised by students

Topic	Description	Num.
ANTLR	Misunderstood grammar design	17
LLVM	Misusage of data structures and methods	32
C++	Unfamiliar with C++ new features	23
Environment	Unfamiliar with Unix, Bash, CMake, Git	6
Framework	Misunderstood the design; Need more freedom	13

6 CONCLUSION AND FUTURE WORK

In the paper, we present our new comiler practice system based on ANTLR and LLVM in order to meet the needs of the Industry. Through performing the new practice system in Fall 2017, the majority of students adapt to the new changes and obtain a great harvest. Although students also encounter many problems about configuring working environments and utilizing tools, as well as not good at writing documents, these problems will be addressed in the future by providing clearer and more precise guidance. Since the projects provided are pluggable, different universities can select part of them to practice.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[2] Thomas Ball and Benjamin Zorn. 2015. Teach Foundational Language Principles. *Commun. ACM* 58, 5 (April 2015), 30–31. DOI: https://doi.org/10.1145/2663342
[3] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04*. Palo Alto, California. http://llvm.org/
[4] Terence Parr and Kathleen Fisher. 2011. LL(*): The Foundation of the ANTLR Parser Generator. In *32nd PLDI*. ACM, New York, NY, USA, 425–436. DOI: https://doi.org/10.1145/1993498.1993548
[5] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *OOPSLA '14*. ACM, New York, NY, USA, 579–598. DOI: https://doi.org/10.1145/2660193.2660202
[6] Alex Radermacher and Gursimran Walia. 2013. Gaps Between Industry Expectations and the Abilities of Graduates. In *44th SIGCSE*. ACM, New York, NY, USA, 525–530. DOI: https://doi.org/10.1145/2445196.2445351
[7] Alex Radermacher, Gursimran Walia, and Dean Knudson. 2014. Investigating the Skill Gap Between Graduating Students and Industry Expectations. In *36th ICSE Companion*. ACM, New York, NY, USA, 291–300. DOI: https://doi.org/10.1145/2591062.2591159
[8] Amirali Sanatinia and Guevara Noubir. 2016. On GitHub's Programming Languages. *CoRR* abs/1603.00431 (2016). arXiv:1603.00431 http://arxiv.org/abs/1603.00431