

How are Students Struggling in Programming? Understanding Learning Processes from Multiple Learning Logs

Yuta Taniguchi

Faculty of Information Science
and Electrical Engineering,
Kyushu University
Fukuoka 819-0395, Japan
taniguchi@ait.kyushu-
u.ac.jp

Fumiya Okubo

Faculty of Business
Administration,
Takachiho University
Tokyo 168-8508, Japan
fokubo@takachiho.ac.jp

Atsushi Shimada

Faculty of Information Science
and Electrical Engineering,
Kyushu University
Fukuoka 819-0395, Japan
atsushi@ait.kyushu-
u.ac.jp

Shin'ichi Konomi

Faculty of Arts and Science,
Kyushu University
Fukuoka 819-0395, Japan
konomi@artsci.kyushu-
u.ac.jp

ABSTRACT

This study discuss how students resolve compilation errors with textbooks in C programming exercises. A student's understanding of the programming language is strongly connected with compilation results. Resolving compilation errors requires one to check source code carefully and understand the language better. Reading textbooks is a typical learning activity of students in resolving errors as well as asking teachers. Therefore, learning processes in a programming exercise course could be understood by combining students' compilation logs and reading logs of e-textbooks. In this paper, we present preliminary results of analysis on students' struggling to resolve errors. Using a dataset collected during a semester in our university, we discuss the universality of errors in terms of students and exercise questions. Furthermore, we reveal the positive and negative impact of reading e-textbooks on error resolutions on a per-page basis.

Keywords

programming exercise, learning process, error resolution

1. INTRODUCTION

Programming techniques are getting more and more attention recent years, and are being introduced into educational curriculum in primary and secondary education as well as higher education. The C programming language is one of the

most important and popular programming language widely used in industries over the past decades. However, there are many obstacles for students in learning the programming language, and thus how to understand and support their learning is an important question.

On the one hand, we need to support students to fix errors in their source code. It is said nearly half of the time and effort are spent in debugging during the development of a program [3]. Park et al. reported that common syntax errors tend to remain in source code for a long time [4]. It requires too much effort for a student to identify and fix such errors in learning.

On the other hand, we have to grasp how students struggle to resolve errors in source code. We can consider that many types of mistakes in source code reflect a student's understanding at some point of his or her learning process. Tracking the resolution of compilation errors, we could understand the learning processes of students better.

To this end, many work analyzed errors in programming courses. Fu et al. [1] have proposed a web-based system that helps teachers to support student during class by providing real-time dashboard to grasp students' learning situations. Their system is helpful to overview the current situation of a single class at a glance. However, the information on the dashboard is somewhat superficial and based on short-period data, and how they struggle to resolve errors is not discussed.

Helminen et al. [2] addressed the process in which students struggle to resolve errors. However, in their study, only limited activities of selecting, ordering, and indenting code fragments are analyzed, and activities such as referring external learning materials are not considered. For understanding students' learning processes, it is significant to know how

students search learning resources for necessary information and acquire knowledges. Nevertheless, only a limited number of studies focused on students' try-and-error and knowledge acquisition in learning processes of programming languages.

In this paper, we analyze how students struggle to resolve compilation errors with course materials in a programming exercise course. Toward this end, we employ both compilation logs and page view logs of e-textbooks, and characterize compilation errors in relation to exercise questions and individual students. Furthermore, our preliminary result shows the positive or negative contribution of every page of course materials in resolution of a particular error.

2. METHOD

2.1 Compilation and e-Book Operation Logs

We focus on the data obtained from the multiple classes of the C programming course of our university offered in the first semester 2017. The course is mainly for freshmen, and includes lectures and coding exercises. There are about 20 classes for the course in a semester, and almost all of the courses are taught by different teachers. We have a set of standard course materials and usually they use it in their teaching, but it is not enforced.

In exercise, we use the compiler “gcc”, from the GNU compiler collection, on a remote Linux server. The compiler program is modified from the original version so that it can record students' learning logs. More precisely, when it is executed, it saves given commandline arguments, the contents of given source files, and the output of the compiler as well as the time and user of the invocation. Since a commandline and source code are available as logs, we can reproduce what a student tried and what he or she obtained as a result.

In most cases, the compiler's output is produced only when there are some problems. The majority of problems are in source code, which result in compilation errors. The others are caused by errors outside source code, such as inadequate arguments and wrong filenames, and they are not recognized as compilation errors. We ignore the latter type of errors in this study since we are not interested in the learning process of compilation itself but in that of a programming language.

We also utilize students' activity data of reading course materials. Our materials are provided on our own e-book system. Students read those materials on the web, and their operations on the system are collected immediately as events. There are variety of operation types including page flipping, full text searching, bookmarking, and so on.

Combined with compilation logs, these event logs tell us how students learned during exercise. For example, after a compilation failure, some students just repeat compilation without necessary modification of source code, and some other students go back to course materials and try to find the key to solve the problem. It might be also possible to evaluate the ability of students. If a student can quickly rewrite source code without looking any materials when a compilation failed, we can consider he or she is well experienced. We should care a student who read many pages of course materials and still failing to compile their source code.

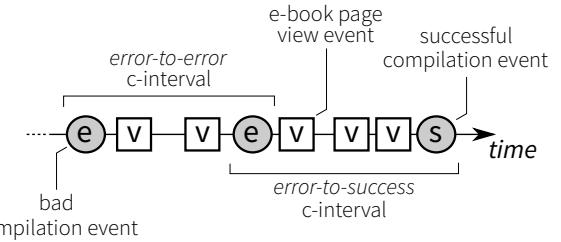


Figure 1: An example of a session and c-intervals.

```
4_a-1.c: In function 'main':  
4_a-1.c:9: error: conflicting types for 'X'  
4_a-1.c:5: note: previous declaration of 'X' was here
```

Figure 2: An example output of compilation error. The first line describes a context where the following error of the second line happened. The final line represents an error, but a note. Only a single error is included in this example output.

2.2 Timeline

We call a sequence of events a *timeline*. A timeline is composed of compilation events and e-book system's page view events, and represents a students' activities in a class. A timeline can be divided into shorter parts by exercise problems that a student was working on at a point of time. In exercise, students are given several problems to solve and instructed to save source code for the problems in separate files with specified filenames. Since a filename and a exercise problem is connected, we can identify a corresponding task from a compilation event log and split timelines into some parts. We call such a part *session* in this paper.

Furthermore, we introduce the concept of *c-interval*. A c-interval is a part of a session which starts and ends with compilation events, and includes only non-compilation events between them. This is a basic unit considered in our analysis because we are interested in what a student does after a compilations and how such activities affect the succeeding compilation.

Seeing if compilation events of a c-intervals are successful or not, we can classify c-intervals into four types: *error-to-error*, *error-to-success*, *success-to-error*, and *success-to-success*. Figure 1 shows an example of a session. In this example, a session consists of eight events; three are compilation events and five are e-book system's page view events. Two types of c-interval, *error-to-error* and *error-to-success* ones, are also shown.

2.3 Compilation Errors

First of all, we normalize the language of error messages. Some of the error messages could be recorded in a non-English language of a student's preference. In our dataset, most of error messages are written in English, but some Japanese or Chinese words are also included. Therefore, we translate such non-English portions of messages into English by a dictionary based method. Please note that the dictionary is currently incomplete and it affects the results shown later in this paper.

```
c_b-1.c: In function 'main':
c_b-1.c:8: error: expected ';' before '}' token
c_b-1.c:9: error: 'else' without a previous 'if'
c_b-1.c:9: error: expected ';' before '}' token
c_b-1.c:10: error: 'else' without a previous 'if'
c_b-1.c:10: error: expected ';' before '}' token
```

Figure 3: An example of compilation output with multiple errors. In this example five error messages are included led by a message describing a context in which those errors happened.

```
{param1}: error: stray '{param2}' in program
{param1}: warning: null character(s) ignored
{param1}: error: expected '{param2}', before '{param3}' token
{param1}: error: expected '{param2}', before '{param3}'
{param1}: error: expected expression before '{param2}', token
{param1}: error: expected expression before '{param2}'
{param1}: error: too few arguments to function '{param2}'
{param1}: error: stray '{param2}', in program
{param1}: error: conflicting types for '{param2}'
{param1}: error: invalid suffix "{param2}" on integer constant
```

Figure 4: Example templates of error messages obtained from our dataset.

We identify every error message in compiler output with our own parser. We developed a parser program that automatically identify error messages in compiler output based on a heuristic algorithm. Figure 2 shows an example output of compilation error. The first line describes a context where the following error of the second line happened. The final line represents an error, but a note. Consequently, our program identifies only a single error in this example. Figure 3 shows another example output of compilation error with multiple errors. In this example 11 error messages are identified by the parser which is led by a message describing a context in which those errors happened.

Many of found error messages share the same underlying structures, for example:

```
4_a-1.c:9: error: conflicting types for 'X'
8_b-2.c:21: error: conflicting types for 'count'
```

Such a structure can be represented as a single template like

```
{p1}:{p2}: error: conflicting types for '{p3}'
```

where {p1}, {p2}, and {p3} are placeholders of the template. We believe such a template represents the essentials of errors better than raw messages. Hence we identify a template text with an error in later analysis.

We investigated actual error messages and found out heuristic rules to obtain a template from a message. Based on the rules, we extracted all the templates from our dataset. There are 52,384 different error messages in our normalized dataset, and they were greatly reduced into 247 message templates. Figure 4 shows a set of examples from them.

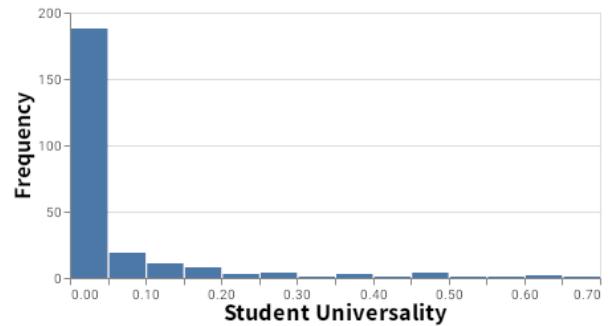


Figure 5: Histogram of student universality of errors. The horizontal axis indicates student universality of errors, and the vertical axis shows the number of errors.

2.4 Analysis

We analyze the universality of errors to know the diversity of students' struggling. Since every student have different understanding, it is expected that there are only a few common errors and many student-specific errors. We also consider the topics of exercise questions influence the tendency of compilation errors. Therefore, we consider two kinds of universality: one based on student and the other based on questions.

Given an error, the former universality can be quantified as the ratio of students out of all students who encountered the error. The latter can be similarly computed as the ratio of questions among all exercise questions where students encountered the error. We call these ratios *student universality* and *question universality* of an error, respectively.

We also analyze how reading material pages impacted on error resolution in students' struggling. To this end, we focus on *error-to-error* and *error-to-success* c-intervals. We assume that all pages viewed during these types of c-intervals were for fixing compilation errors, especially errors found in the earlier compilation event of c-intervals. In the case of *error-to-success*, we consider pages contributed positively in error resolution; contrastingly, we consider pages worked negatively in the case of *error-to-error*.

The contribution of every page are measured for each error as follows. We count how many times a page p_i contributed positively or negatively for the resolution of an error e_j . Given p_i and e_j , let $C_{\text{pos}}^{i,j}$ and $C_{\text{neg}}^{i,j}$ be the numbers of positive or negative cases, respectively. The contribution of p_i for e_j is defined as follows:

$$\text{Contribution}(p_i, e_j) = \frac{C_{\text{pos}}^{i,j} - C_{\text{neg}}^{i,j}}{C_{\text{pos}}^{i,j} + C_{\text{neg}}^{i,j}}$$

With this formulation, contributions are represented by values in $[-1, 1]$. A positive value indicates more positive contributions than negative ones, and vice versa.

3. EXPERIMENT

Figure 5 shows the distribution of the student universality of errors as a histogram. The horizontal axis indicates the student universality, and the vertical axis shows the frequency

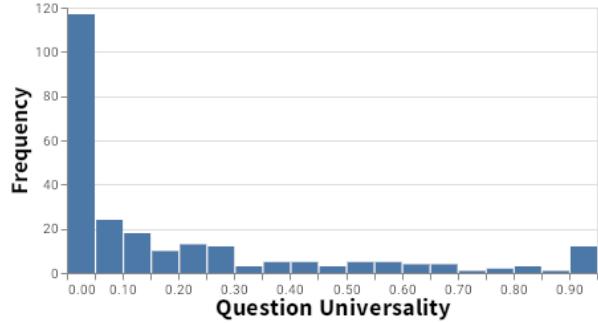


Figure 6: Histogram of the question universality of errors. The horizontal axis indicates the question universality of errors, and the vertical axis shows the frequency of errors.

of errors. From the figure, we can say that there is a small number of common errors, while most of the errors occur to a limited number of students. To be precise, about the three quarters of the errors were encountered by at most five percent of students, and about the half of the errors were encountered by at most one percent of students. Moreover, the common errors that a majority of students encountered were limited to only five particular types.

Figure 6 shows the distribution of the question universality of errors as a histogram. The horizontal axis indicates the universality, and the vertical axis shows the frequency of errors. The figure shows there are more than ten errors that are not problem-specific. It seems that most errors are associated with a few exercise problems. However, compared to that of student universality, there are more errors which occurs for more than half of questions.

Figure 7 is the heatmap that shows contributions of material pages to error resolutions. Each row corresponds to an error, and each column represents a page of a material. Pages are sorted by the order of material usage and then by page numbers. In the heatmap, a cell with positive contribution value is colored reddish. In the reverse case, a cell is colored bluish.

From the figure we can observe positive and negative cases are clearly separated in fairly many cases. This fact suggests that pages does matter for resolving compilation errors. It also seems that students tend to read many useless pages when they struggled to fix errors. Consequently, it may be helpful for teachers to teach student how to understand error messages and find effective material pages.

4. CONCLUSIONS

In this study, we investigated how students struggle to resolve compilation errors with textbooks during exercises, and we employed compilation logs and browsing history of e-textbooks to this end. As the preliminary results, we found that most of the errors are student- and question-specific, and errors universally observable are quite limited. Reading course materials seems to be helpful for error resolution though it is highly dependent on a type of errors. Hence, we conclude that students have different situations individually

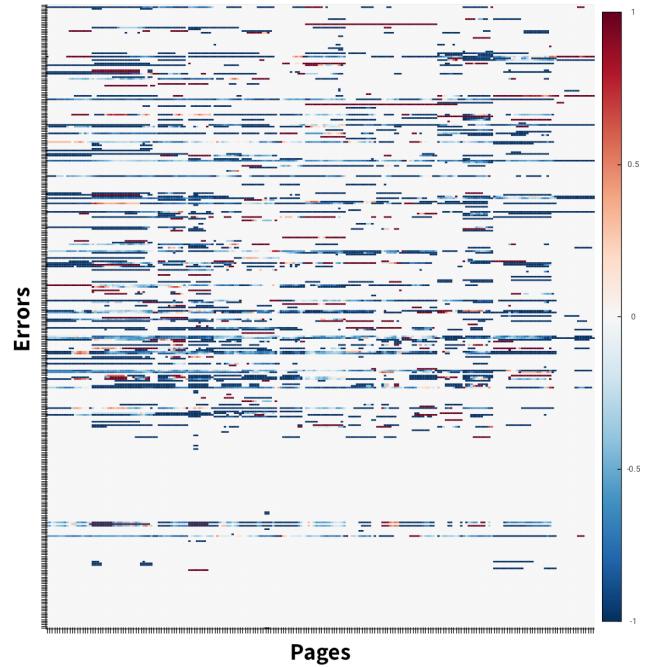


Figure 7: Heatmap showing contribution of material pages to error resolutions. Each row corresponds to an error, and each column represents a page of a material. Reddish color represents positive contribution values, and bluish color represents negative values.

and they have to find helpful material pages to resolve particular errors. This suggests the need of personalized analysis and support in programming education. The limitation of the work includes the lack of student-wise and statistical analyses. We are going to do more personal analysis with more data through several semesters as future work.

5. ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number JP17K12804.

6. REFERENCES

- [1] X. Fu, A. Shimada, H. Ogata, Y. Taniguchi, and D. Suehiro. Real-time learning analytics for c programming language courses. In *Proceedings of the Seventh International Learning Analytics & Knowledge Conference, LAK '17*, pages 280–288, New York, NY, USA, 2017. ACM.
- [2] J. Helminen, P. Ihantola, V. Karavirta, and L. Malmi. How do students solve parsons programming problems?: An analysis of interaction traces. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research, ICER '12*, pages 119–126, New York, NY, USA, 2012. ACM.
- [3] G. J. Myers. A controlled experiment in program testing and code walkthroughs/inspections. *Commun. ACM*, 21(9):760–768, Sept. 1978.
- [4] T. H. Park, B. Dorn, and A. Forte. An analysis of html and css syntax errors in a web development course. *Trans. Comput. Educ.*, 15(1):4:1–4:21, Mar. 2015.