# Investigating the Difficulty of Commercial-level Compiler Warning Messages for Novice Programmers

Yoshitaka Kojima[1], Yoshitaka Arahori[2] and Katsuhiko Gondow[2]
[1]*formerly Department of Computing Science, Tokyo Institute of Technology, Tokyo, Japan*
[2]*Department of Computing Science, Tokyo Institute of Technology, Tokyo, Japan*

Keywords: Programming Education, Commercial-level Compiler, Compiler Warning Messages, Novice Programmer, Sample Code Set.

Abstract: Many researchers refer to the folklore "warning messages in commercial-level compilers like GCC are difficult for novice programmers, which leads to low learning efficiency." However, there is little quantitative investigation about this, so it is still unknown if (and to what extent) the warning messages are really difficult. In this paper, we provide a quantitative investigation about the difficulty of the warning messages. More specifically, as a sample code set we first collected 90 small programs in C language that are error-prone for novice programmers. Then we performed the investigation on the warning emission and its difficulty for 4 compilers and 5 static analysis tools, which are all commercial-level, using the sample code set. The difficulty of warning messages were evaluated by 7 students as research participants, using 4 evaluation criteria of clarity, specificity, constructive guidance, and plain terminology. As a result, we obtained several important quantitative findings: e.g., the deviation of warning emission presence in compilers and static analysis tools is large; and 35.7% of warning messages lack clarity, and 35.9% of warning messages lack specificity, which suggests roughly one third of warning messages are difficult for novice programmers to understand.

## 1 INTRODUCTION

Many researchers refer to the folklore and experience "warning messages in commercial-level compilers like GCC are difficult for novice programmers, which leads to low learning efficiency." (Pears et al., 2007; Nienaltowski et al., 2008; Marceau et al., 2011b; Marceau et al., 2011a; Traver, 2010)

For example, for the code fragment `if(a==2&b==4)` in the C programming language, where `&` (bitwise-and operator) is misused instead of `&&` (logical-and operator), GCC-4.7.2 emits the message:

> warning: suggest parentheses around comparison in operand of '&'

Since `==` has the higher precedence than `&` in C, GCC interprets the code fragment as `if((a==2)&(b==4))` and the message suggests to modify it to `if(a==(2&b)==4)`. But this warning message is very difficult for novice programmers, since this modification does not solve the problem, and it is difficult for novice programmers to imagine that the warning message points out the possibility of `&`'s precedence problem. This is a false positive of GCC warning mechanism that suggests to modify, for example, `if(x&0xFF00==0)` to `if((x&0xFF00)==0)`, where the message becomes correct.

However, as far as we know, there is little quantitative investigation about this (see Sec. 2), so it is still unknown if (and to what extent) the warning messages are really difficult. In this paper, we provide a quantitative investigation about the difficulty of the warning messages. This kind of investigation is crucial in the following points:

- The result can be used as a comparison or benchmark to research and develop better compiler messages.
- The result can also be used for programming instructors to select more appropriate compilers for their students.

Moreover, this kind of investigation is not trivial in the following points:

- There is no sample code set including small error-prone programs for novice programmers. Thus, it is necessary to first build such a sample code set.

Kojima Y., Arahori Y. and Gondow K..
Investigating the Difficulty of Commercial-level Compiler Warning Messages for Novice Programmers.
DOI: 10.5220/0005437404830490
In *Proceedings of the 7th International Conference on Computer Supported Education* (CSEDU-2015), pages 483-490
ISBN: 978-989-758-108-3

- It is impossible to provide an absolute criterion of the difficulty. Thus, the investigation is essentially based on subjective judgment, and the criteria of difficulty varies to some extent among investigation participants. Moreover, the investigation itself is a big burden on the participants, since they have to carefully read many programs and warning messages. These issues make it challenging to design the investigation.

In this paper, we provide a quantitative investigation about the difficulty of C compiler warning messages using the following steps.

- Step 1: As a sample code set, we first collected 90 small programs that are error-prone for novice programmers, mainly including semantic errors and logical errors (Sec. 3.2).
- Step 2: We then obtained all the warning messages for the above sample code set by 4 compilers and 5 static analysis tools which are all commercial-level.
- Step 3: The difficulty (or effectiveness) of warning messages were evaluated by 7 students as research participants[1], using 4 evaluation criteria of clarity, specificity, constructive guidance, and plain terminology (Sec. 3.4).

The main contributions of this research is as follows:

- We have provided the first sample code set in C language that are error-prone for novice programmers, and can be used to measure the difficulty of compiler warning messages.
- We obtained several important quantitative findings: e.g., the deviation of warning emission presence in compilers and static analysis tools is large; and 35.7% of warning messages lack clarity, and 35.9% of warning messages lack specificity, which suggests roughly one third of warning messages are difficult for novice programmers to understand.

## 2 RELATED WORK

There are several papers on compiler messages, summarized in this section. To our knowledge, however, none of them quantitatively investigate the difficulty of warning messages of commercial-level compilers. Thus, this paper is a first trial towards the quantitative investigation.

Nienaltowski et al. studied the effect of different compiler message styles (short, long, visual form) on how well and quickly students identify program errors (Nienaltowski et al., 2008). The students were asked to answer the cause of the error for 9 multiple choice questions. The aim of this study is not to explore the difficulty of warning messages, and only 9 questions were used, while in our study 1,296 questions for 90 small programs are used.

Marceau et al. pointed out that there are few rigorous human-factors evaluation on compiler messages (Marceau et al., 2011b), and they investigated the effectiveness of compiler messages of DrRacket (Marceau et al., 2011b; Marceau et al., 2011a) DrRacket is not a commercial-level compiler, but a programming environment for novice programmers.

Jackson et al. identified common Java errors for novice programmers using their automated error collection system (Jackson et al., 2005). Kummerfeld and Kay proposed a novel method to help novice programmers better understand the error messages, by providing a Web-based reference guide that catalogues common incorrect programs, compiler error messages for them, error explanations, and possible corrections (Kummerfeld and Kay, 2003). Dy and Rodrigo proposed a detection tool that checks novice student code for non-literal errors (i.e., compiler-reported errors that do not match the actual error) and produces more informative error reports (Dy and Rodrigo, 2010). All these papers focused on simple syntax errors like an undefined variable, and did not investigate the difficulty of compiler messages, while our research investigate the difficulty of compiler messages for semantic and logic errors.

BlueJ (Kölling et al., 2003)Expresso (Hristova et al., 2003) Gauntlet (Flowers et al., 2004) are programming environments for novice programmers that aim to provide more understandable error messages. Gross and Powers surveyed programming environments for novice programmers such as BlueJ (Gross and Powers, 2005). None of these papers mentioned or compared with the error messages of commercial-level compilers like GCC.

## 3 INVESTIGATION METHOD

### 3.1 Purpose and Outline of the Investigation

The purpose of our research is to provide a quantita-

[1]Four 4th-year undergraduates and three 1st-year graduates in Dept. of Computer Science of Tokyo Institute of Technology. They are all members of author's laboratory, and they took programming exercises in C, Scheme and Java through the lectures.

Table 1: Compilers and static analysis tools used in investigation.

| Abbrev. | Compiler names |
|---|---|
| GCC | GCC-4.7.2 |
| VS | Microsoft Visual Studio Express 2012 for Windows Desktop (Visual C++ Compiler) |
| Clang | Clang-4.2.1 |
| ICC | Intel C++ Studio XE 2013 for Linux |
| **Abbrev.** | **Static analysis tool names** |
| Eclipse | Eclipse CDT: Juno Service Release 1 |
| VS-SA | VS Code Analysis |
| C-SA | Clang Static Analyzer 269 |
| ICC-SA | ICC Static Analysis |
| Splint | Splint-3.1.2 (22 Sep 2012) |

tive investigation about the difficulty of the warning messages of commercial-level compilers and static analysis tools. More specifically:

- As targets of our investigation, we selected 4 compilers and 5 static analysis tools listed in Table. 1 (the compilers/tools for short), which are all widely used, easily available and of commercial-level quality.
- As a sample code set, we collected 90 small programs that are error-prone for novice programmers, mainly including semantic errors and logical errors (Sec. 3.2).
- We obtained all the warning messages that the compilers/tools emitted for the 90 error-prone programs. Then, we analyzed their emission rates and deviation.
- We investigated, by the questionnaire of 1,264 questions (4 evaluation criteria for the total 316 warning messages), to what extent and how the warning messages are difficult for novice programmers. The 90 error-prone programs and 316 warning messages were given to 7 research participants, and then they answered the questions using 4 evaluation criteria of clarity, specificity, constructive guidance, and plain terminology (Sec. 3.4), allowing subjective judgment to some extent.

In our investigation, we used the compiler options that emit as many warning messages as possible, except the options for optimization. For example, we used the GCC option: `-Wall -Wextra -pedantic -Wfloat-equal`. This is because we wanted to know the upper limit of the power that the compilers/tools emit warning messages.

```
#include <stdio.h>
int main (void) {
    int n = 15;
    if (1 <= n <= 10)
        printf ("1 <= %d <= 10\n", n);
}
```

Figure 1: An example of logic errors: the programmer's intention is `(1<=n && n<=10)`.

Table 2: Error categories and their numbers of the collected small sample programs

| error category | # of programs |
|---|---|
| pointer/array | 31 |
| conditional | 16 |
| function | 16 |
| variable | 14 |
| expression/statement | 13 |
| total | 90 |

## 3.2 Error-prone Programs for Novice Programmers

Although the level of "novice" varies, we define "novice" as a programmer who can correct syntax errors somehow, but is not good at correcting semantic errors and logic errors. This reason is twofold. First, the related work (Jackson et al., 2005; Kummerfeld and Kay, 2003; Dy and Rodrigo, 2010) mainly dealt with syntax errors, but did not dealt with semantic errors and logic errors. Second, in our observation, semantic errors and logic errors are far more difficult for novice programmers than syntax errors.

Here we use the term "semantic error" as a program that is semantically incorrect and causes a warning message such as division by zero, type mismatch, undefined behavior, and unspecified behavior. Note that from "semantic errors" we exclude some explicit errors at compile-time such as doubly defined variables, since they are relatively easy for novice programmers.

Also we use the term "logic error" as a program that is correct syntactically and semantically, but contrary to the programmer's intention. Fig. 1 shows an example of logic errors; the operator `<=` is left-associative, so the conditional expression in Fig. 1 is equal to `((1<=n)<=10)`, whose result value becomes always true since the result of `(1<=n)` is `0` or `1`. This is apparently contrary to the programmer's intention, which is probably `(1<=n && n<=10)`.

## 3.3 Collecting Small Sample Programs

As a sample code set, we collected 90 small programs in C language that have semantic errors or logic errors mentioned in Sec. 3.2; all of them are error-prone for

```
#include <stdio.h>
int main (void) {
    int a = 2, b = 4;
    if (a == 2 & b == 4)
        printf("a = 2 and b = 4\n");
}
```

Figure 2: Example of mistakes where bitwise-and `&` is misused instead of logical-and `&&`.

novice programmers. This section describes how to collect them.

To cover the various types of error-prone programs, we thoroughly investigate 8 Web programming forums[2] and C FAQ[3], and then collected 90 error-prone programs from there. All of them are small with around 10 lines of code. Fig. 1 is an example of the collected programs. Table. 2 shows the error categories and their numbers of the collected programs[4].

Since the programmer's intentions are not obvious only from the programs, we simply provided the programmer's intentions to the research participants. For Fig. 1, for example, "Error description: `1<=n<=10` is a mathematical comparison notation; Solution: change it to `(1<=n)&&(n<=10)`" in Japanese was given.

Through this collecting activity, we obtained the following valuable findings:

- The collecting activity was very tedious and time-consuming, since the Web forums that we used have a lot of similar redundant or unrelated questions to our research purpose (eg. questions about syntax errors, coding styles and API usage).
- The sample code set attached to Clang was useless for our purpose, since it aims to help compiler writers, not novice programmers.

## 3.4 Evaluation Criteria

We selected the following 4 criteria to evaluate warning messages, which have been proposed in the previous work (Traver, 2010; Horning, 1976).

- Clarity: Does the message clearly tell what is the problem?

[2]stackoverflow, GIDForums, Tek-Tips Forums, http://bytes.com, http://www.cprogramming.com, http://dixq.net/, http://chiebukuro.yahoo.co.jp/dir/list/d2078297650, http://oshiete.goo.ne.jp/category/250, http://www.ncos.co.jp/products/cgi-bin/errorcall.cgi (the last 4 forums are in Japanese only)

[3]http:// c-faq.com/

[4]The collected programs are accessible at (Gondow, 2015).

```
#include <stdio.h>
int main (void) {
    int i = 0;
    scanf("%d", i); printf("%d\n", i);
}
```

Figure 3: Example of mistakes where `&` is wrongly omitted in the `scanf` parameter.

- Specificity: Does the message provide a specific information to identify the problem?
- Constructive guidance: Does the message provides a guidance or hint to correct the problem?
- Plain terminology [5]: Does the message only use plain technical terms?

Note that these criteria depends on each other. For example, Constructive guidance depends on Clarity, since if the compiler wrongly identifies a logical error, the consequent guidance will also be wrong.

The previous work (Traver, 2010; Horning, 1976) proposed other criteria like Context-insensitivity, Consistency, Locality, which are not used in our investigation. Context-insensitivity means a compiler should emit the same message for the same error regardless of the context; Consistency means the terminology or representation of messages should be consistent; and Locality means that a compiler should indicate the error place near the true origin of the error. These criteria are not appropriate to our research, since all of these criteria require a much larger scale investigation (eg. larger programs), but the collected sample programs are all small.

We use the 4 criteria we selected as follows:

- Grading: We ask the research participants to evaluate warning messages in three grades as listed in Table. 3. This is because the evaluation is based on a subjective judgment, which makes finer grading difficult.
- Clarity, Constructive Guidance: Fig. 2 is an example program for that GCC emits very difficult warning message (mentioned in Sec. 1), where bitwise-and `&` is misused instead of logical-and `&&`. The GCC's warning message for Fig. 2 is

  > warning: suggest parentheses around comparison in operand of '&'.

  First, the message lacks Clarity (judged as C), since the message does not tell that the problem is the misuse of bitwise-and `&` instead of logical-and `&&`, although it is difficult for compilers to know that the programmer's intent is logical-and

[5]The term "programmer language" is used in (Traver, 2010).

Table 3: Meaning of grading of evaluation criteria.

| | A | B | C |
|---|---|---|---|
| Clarity | clear | a little bit unclear | unclear |
| Specificity | sufficient | a little bit insufficient | insufficient |
| Constructive guidance | given | not given | wrongly given |
| Plain terminology | understandable | not understandable | compiler-dependent representation |

`&&`. Second, Constructive guidance for the message is wrongly given (judged as C), since the message suggests the use of parentheses, but this modification does not solve the problem.

- Specificity: Fig. 3 is an example that Specificity is a little bit insufficient, where the address operator `&` is missing just before the argument `i` of the call `scanf`. The ICC's warning message for Fig. 3 is

  > warning: argument is incompatible with corresponding format string conversion.

  In this message, there is some specific information about the error that the cause comes from the incompatibility between the argument type and the conversion in the format string. But there is no specific information that `scanf` requires a pointer type, argument in the message is `i`, and corresponding format string conversion is `%d`. Thus, Specificity is a little bit insufficient in the warning message (judged as B).
- Plain terminology: For example, some technical terms in the C standards like "unspecified behavior" and "sequence point" are too difficult for novice programmers (judged as B). For another example, if the message has the size information of 40 bytes for the array definition `int a[10];`, the message has compiler-dependent representation, since the size of `int` in C is compiler-dependent (judged as C).

# 4 RESULT OF INVESTIGATION

## 4.1 Deviation of Warning Message Output

Table. 4 summarizes the numbers of sample programs that the compilers/tools emitted warning messages for[6]. For the total 90 sample programs, VS-SA emitted the smallest 21 warning messages, while Splint emitted the largest 69 ones.

[6]Raw data, such as all collected warning messages, is accessible at (Gondow, 2015).

Table 4: The numbers of sample programs that the compilers/tools emitted warning messages for.

| Compilers | # detected | Tools | # detected |
|---|---|---|---|
| GCC | 45 | Eclipse | 39 |
| VS | 37 | VS-SA | 21 |
| Clang | 38 | C-SA | 43 |
| ICC | 36 | ICC-SA | 61 |
| | | Splint | 69 |

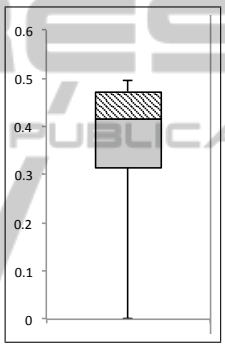


Figure 4: Box plot of standard deviation of the presence of warning messages.

### 4.1.1 Deviation is Large

The result indicates that the deviation of the presence or absence of warning message output for the same sample program is large among the compilers/tools.

Table. 5 shows the frequency distribution of the numbers of the compilers/tools that emitted warning messages for each sample program. For example, there are only 16 sample programs out of 90 that all 9 compilers/tools emitted warning messages for. On the other hand, there are 13 sample programs out of 90 that only 3 compilers/tools emitted warning messages for.

By quantifying the presence of a warning message as 1 and the absence as 0, we obtain 9 numerical values (each 0 or 1) for the 9 compilers/tools and one sample program. Fig. 4 is the box plot of the standard deviation of this 9 numerical values for all 90 sample programs. The median is 0.416, and the arithmetic mean is 0.325, which indicates that the deviation of warning message output is large.

Table 5: The frequency distribution table of the numbers of the compilers/tools that emitted warning messages for each sample program.

| # compilers/tools that emitted warming messages | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # sample programs (frequency) | 4 | 12 | 8 | 13 | 9 | 8 | 5 | 5 | 10 | 16 | 90 |

```
#include <stdio.h>
#include <string.h>
int main(void) {
    char *p = "Hello";
    // write to string literal
    strcat(p, "World");
    printf("%s\n", p);
}
```

Figure 5: Example of programs that only 3 compilers/tools emitted warning messages.

```
#include <stdio.h>
#include <string.h>
int main(void) {
    char from[] = "Hello";
    char *to;
    // write through uninitialized pointer
    strcpy(to, from);
    printf("%s\n", to);
}
```

Figure 6: Example of programs that all 9 compilers/tools emitted warning messages.

#### 4.1.2 Example of Deviation

Fig. 5 is an example of program that only 3 compilers/tools emitted warning messages for. The call `strcat` in Fig. 5 attempts to write to string literal that are not writable in the C language. For the program in Fig. 5, only C-SA, Splint and ICC-SA emitted warning messages.

Fig. 6 is an example of program that all 9 compilers/tools emitted warning messages for. It is interesting that the numbers of compilers/tools that emitted warning messages for Fig. 5 and Fig. 6 are quite different (3 vs. 9), but the static analysis required to emit the warning messages are mostly the same for both of Fig. 5 and Fig. 6.

### 4.2 Difficulty of the Warning Messages for Novices

For the 90 sample programs, we performed the investigation of warning message difficulty of the compilers/tools (Sec. 3.1) by 7 students as research participants, using 4 evaluation criteria (Sec. 3.4). The result of questionnaire[8] is summarized in Table. 6 and Table. 7. Table. 6 shows the frequency distribution by the compilers/tools, while Table. 7 shows the one by the research participants.

```
#include <stdio.h>
int main(int argc, char **argv) {
    while (argv++ != NULL)
        printf("%s\n", *argv);
}
```

```
main.c:5:24: Possible out-of-bounds read: *argv
  Unable to resolve constraint:
  requires maxRead(argv @ main.c:4:12) >= 1
    needed to satisfy precondition:
  requires maxRead(argv @ main.c:5:25) >= 0
A memory read references memory beyond the allocated storage.
```

Figure 7: Example of warning messages by Splint.

In Table. 6, the frequency of the GCC's Clarity 'A' is 186, which means that for the GCC's 45 warning messages, the 7 research participants judged the total 186 warning messages as 'A'. On the other hand, in Table. 7, the frequency of Clarity 'A' of the research participant ID '0' is 137, which means that the research participant ID '0' judged 137 warning messages as 'A' out of the total 316 ones,

Major findings from this result are as follows:

- Table. 6: The majority of the result of Clarity, Specificity and Plain Terminology is 'A'. However, 35.7%[9] warning messages lack Clarity, and 35.9%[10] warning messages lack Specificity[11]. Roughly speaking, this result quantitatively indicates one third of warning messages are difficult for novice programmers to understand.
- Table. 6: The numbers of 'A' and 'C' in Constructive Guidance are small, which means there are a few guidance of helpful (A) or wrong (C). This probably indicates that the present commercial-level compilers/tools are negative for emitting helpful guidance not to increase wrong ones (false

[7] Eclipse, for example, sometimes reuses the underlying GCC's warning messages, which are excluded in this table. Thus, # detected in Table. 6 and Table. 7 are less than ones in Table. 4.

[8] Anonymized raw data, such as the result of questionnaire, is accessible at (Gondow, 2015).

[9] $35.7\% = (577+212) \times 100/(1423+577+212)$

[10] $35.9\% = (574+220) \times 100/(1418+574+220)$

[11] 24% messages lack both Clarity and Specificity.

Table 6: The frequency distribution table of evaluation by compilers/tools.

| Compilers /Tools | # detected[7] | Clarity | | | Specificity | | | Constr. Guidance | | | Plain Term. | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | A | B | C | A | B | C | A | B | C | A | B | C |
| GCC | 45 | 186 | 77 | 52 | 181 | 88 | 46 | 19 | 260 | 36 | 297 | 16 | 2 |
| VS | 37 | 168 | 64 | 27 | 168 | 58 | 33 | 30 | 217 | 12 | 243 | 10 | 6 |
| Clang | 38 | 187 | 60 | 19 | 195 | 46 | 25 | 55 | 199 | 12 | 250 | 13 | 3 |
| ICC | 36 | 141 | 85 | 26 | 135 | 94 | 23 | 19 | 226 | 7 | 242 | 10 | 0 |
| Eclipse | 8 | 24 | 26 | 6 | 24 | 26 | 6 | 2 | 54 | 0 | 55 | 1 | 0 |
| VS-SA | 21 | 102 | 33 | 12 | 108 | 30 | 9 | 20 | 125 | 2 | 109 | 19 | 19 |
| C-SA | 20 | 118 | 20 | 2 | 98 | 37 | 5 | 20 | 120 | 0 | 133 | 6 | 1 |
| ICC-SA | 42 | 209 | 65 | 20 | 179 | 87 | 28 | 35 | 255 | 4 | 280 | 10 | 4 |
| Splint | 69 | 288 | 147 | 48 | 330 | 108 | 45 | 64 | 383 | 36 | 348 | 97 | 38 |
| Total | 316 | 1423 | 577 | 212 | 1418 | 574 | 220 | 264 | 1839 | 109 | 1957 | 182 | 73 |

Table 7: The frequency distribution table of evaluation by research participants.

| Research Participant ID | # detected[7] | Clarity | | | Specificity | | | Constr. Guidance | | | Plain Term. | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | A | B | C | A | B | C | A | B | C | A | B | C |
| 0 | 316 | 137 | 131 | 48 | 127 | 146 | 43 | 24 | 289 | 3 | 218 | 86 | 12 |
| 1 | 316 | 198 | 86 | 32 | 189 | 103 | 24 | 19 | 278 | 19 | 288 | 14 | 14 |
| 2 | 316 | 263 | 44 | 9 | 239 | 56 | 21 | 16 | 290 | 10 | 292 | 5 | 19 |
| 3 | 316 | 181 | 88 | 47 | 201 | 63 | 52 | 31 | 271 | 14 | 298 | 15 | 3 |
| 4 | 316 | 248 | 49 | 19 | 221 | 75 | 20 | 139 | 149 | 28 | 286 | 23 | 7 |
| 5 | 316 | 212 | 60 | 44 | 166 | 97 | 53 | 17 | 284 | 15 | 292 | 22 | 2 |
| 6 | 316 | 184 | 119 | 13 | 275 | 34 | 7 | 18 | 278 | 20 | 283 | 17 | 16 |
| Total | 2212 | 1423 | 577 | 212 | 1418 | 574 | 220 | 264 | 1839 | 109 | 1957 | 182 | 73 |



positives).

- Table. 6: The result of questionnaire for Plain Terminology in Splint is bad; 'A' is 72.0% for Splint while 'A' is 93.1% on average for the others. Fig. 7 shows an example of difficult messages by Splint. `maxRead` is a Splint-specific terminology, denoting the highest index of a buffer that can be safely used as rvalue. Some novice programmers may understand the message indicates a possible buffer overrun, but almost cannot understand how Splint inferred in terms of `maxRead`. In our observation, this is because Splint attempts to emit more helpful, precise and descriptive messages for the programs that the other compilers/tools do not. If this assumption is correct, this suggests that it is challenging to improve the understandability of warning messages for novice programmers only using plain terminology.
- Table. 7: The result of questionnaire by the research participants seems to have some similar tendency, but they are significantly different for each of 4 criteria. For example, $\chi^2(12) = 191.7 p = 1.71 \times 10^{-34} < 0.05$ for Clarity, where the null hypothesis is that participants and their judgments for Clarity are independent, and it is rejected. This is, however, obviously because they include subjective judgment.

# 5 TOWARDS USABLE COMPILER FOR NOVICE PROGRAMMERS

It is very important but essentially difficult to improve compiler warning messages. There are several reasons for this. It is quite difficult to automatically obtain the programmer's intention from a program. Even worse, Nienaltowski's study (Nienaltowski et al., 2008) suggests more detailed messages do not help the participant's performance (this is the reason why we did not use compiler's verbose options). Kummerfeld's method (Kummerfeld and Kay, 2003) that catalogues compiler error messages and possible corrections might be effective, but its maintenance cost is very high since the catalogue must be updated whenever new compilers are released.

One idea to improve compiler messages is to replace a bad warning message by a good one of other compilers. Different compilers emit different (i.e. good and bad) messages for the same program. By this, accidental but not essential bad messages can be improved. Another idea is to incorporate heuristics or knowledge into compilers that tells how novice programmers make mistakes. For example, novice programmers are likely to misuse `&` instead of `&&` (Fig. 2).

# 6 CONCLUSION

In this paper, we provided a quantitative investigation about the difficulty of the warning messages. As a result, we obtained several important quantitative findings, which suggests roughly one third of warning messages are difficult for novice programmers to understand, so far as this investigation is concerned.

As future work, we would like to perform the investigation with more participants enough to be statistically significant. Also we would like to explore how to improve bad compiler messages effectively and efficiently, and how to apply it to the education for novice programmers.

# REFERENCES


Dy, T. and Rodrigo, M. M. (2010). A detector for non-literal java errors. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 118–122, New York, NY, USA. ACM.

Flowers, T., Carver, C., and Jackson, J. (2004). Empowering students and building confidence in novice programmers through gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, pages T3H/10–T3H/13 Vol. 1.

Gondow, K. (2015). Sample code set, all compiler warning messages and anonymized result of questionnaire. http://www.sde.cs.titech.ac.jp/cm/.

Gross, P. and Powers, K. (2005). Evaluating assessments of novice programming environments. In *Proceedings of the First International Workshop on Computing Education Research*, ICER '05, pages 99–110, New York, NY, USA. ACM.

Horning, J. J. (1976). What the compiler should tell the user. In *Compiler Construction, An Advanced Course, 2Nd Ed.*, pages 525–548, London, UK, UK. Springer-Verlag.

Hristova, M., Misra, A., Rutter, M., and Mercuri, R. (2003). Identifying and correcting java programming errors for introductory computer science students. *SIGCSE Bull.*, 35(1):153–156.

Jackson, J., Cobb, M., and Carver, C. (2005). Identifying top java errors for novice programmers. In *Frontiers in Education, 2005. FIE '05. Proceedings 35th Annual Conference*, pages T4C–T4C.

Kölling, M., Quig, B., Patterson, A., and Rosenberg, J. (2003). The bluej system and its pedagogy. *Computer Science Education*, 13(4):249–268.

Kummerfeld, S. K. and Kay, J. (2003). The neglected battle fields of syntax errors. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20*, ACE '03, pages 105–111, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Marceau, G., Fisler, K., and Krishnamurthi, S. (2011a). Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 499–504, New York, NY, USA. ACM.

Marceau, G., Fisler, K., and Krishnamurthi, S. (2011b). Mind your language: On novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, pages 3–18, New York, NY, USA. ACM.

Nienaltowski, M.-H., Pedroni, M., and Meyer, B. (2008). Compiler error messages: What can help novices? In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 168–172, New York, NY, USA. ACM.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., and Paterson, J. (2007). A survey of literature on the teaching of introductory programming. *SIGCSE Bull.*, 39(4):204–223.

Traver, V. J. (2010). On compiler error messages: What they say and what they mean. *Adv. in Hum.-Comp. Int.*, 2010:3:1–3:26.