View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by British Columbia's network of post-secondary digital repositories

Examining the Effects of Enhanced Compilers on Student Productivity

by

Devon Harker

B.A., University of Northern British Columbia, 2015

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

UNIVERSITY OF NORTHERN BRITISH COLUMBIA

December 2017

© Devon Harker, 2017

Abstract

Programs written by novices programmers often contain errors. Previous work shows students struggle when compiler error messages are inaccurate, misleading, or both. Loss of productivity caused by poor error messages has not been thoroughly explored in the literature. This thesis examines how enhanced compilers improve the experiences of those learning to program.

The thesis follows fifty non-CS majors with little programming experience through a one-semester CS1-like course at the University of Northern British Columbia, a small western Canadian university. Half of the participants used the enhanced compiler for Java named *Decaf* while the other half used the standard Java compiler. The evidence shows that *Decaf* is beneficial with regards to the number and types of errors generated, productivity, frustration, and confidence in programming ability, and compares results with the literature.

TABLE OF CONTENTS

Abstract ii

Table of Contents iii

List of Figures vi

List of Tables vi

1 Introduction 1

2 Literature Review 6
- 2.1 Alternatives To Compiler Error Messages . . . 7
- 2.2 Investigating Student Learning and Errors . . . 9
- 2.3 Enhanced Compilers . . . 18
- 2.4 Non-Compiler Tools . . . 24

3 Problem Statement 27
- 3.1 What Problem Needs Solving? . . . 27
- 3.2 Why Is This Worth Investigating? . . . 28
- 3.3 Definition of Productivity . . . 29

4 Research Methodology 31
- 4.1 Participant Selection and Grouping . . . 32
- 4.2 Laboratory Computer Setup . . . 34
- 4.3 Programming Pre-assessment and Anonymous Identifiers . . . 35
- 4.4 Consent Form and Information Letter . . . 37
- 4.5 Withdrawal Form . . . 37
- 4.6 Weekly Assignments . . . 38
- 4.7 Questionnaire 1 . . . 38
- 4.8 Questionnaire 2 . . . 39
- 4.9 Laboratory Quiz . . . 40
- 4.10 Questionnaire 3 . . . 40
- 4.11 Anonymous Identifier Corrections . . . 41
- 4.12 Participant Grade Scaling . . . 41
- 4.13 Database Preparation . . . 42

4.14 Laboratory Quiz Snapshot Examination 43
4.15 Statistical Analysis 43
4.15.1 Choice of α 44
4.15.2 When Is An Error Considered To Be Successfully Fixed? . . . 44
4.15.2.1 Which Responses To Error Were Productive and Which Were Unproductive? 46
4.15.3 What Are The Phases Of Compilation? 46
4.15.4 Why Is Timing Data Only Available For The Laboratory Quiz? 50
4.15.5 Q1 — Time and Compilations Per Program 51
4.15.6 Q2 — Productivity 52
4.15.7 Q3 — Phases of Compilation 52
4.15.8 Q4 — Frustration When Fixing Errors 53
4.15.9 Q5 — Confidence in Programming Ability 54
4.15.10 Q6 — Compiler Appreciation 54
4.15.11 Q7 — Self-assessed Versus Measured PPE 55
4.15.12 Q8 — Participant Performance 55
4.15.13 Research Question Which Was Considered But Not Used . . 56

5 Statistical Analysis and Results **57**
5.1 Questionnaires and Programming Pre-assessment 57
5.1.1 Notes On Collected Data 58
5.1.2 Statistical Test Results 58
5.2 Decaf Snapshot Analysis 59
5.2.1 Compilation Error Distribution Analysis 60
5.2.1.1 Assignments 60
5.2.1.2 Laboratory Quiz 61
5.2.2 Timing Data Analysis 61
5.2.3 Performance Analysis 62

6 Discussion **64**
6.1 Q1 — Time and Compilations Per Program 64
6.1.1 Analysis of Compilations 65
6.1.2 Analysis of Time 65
6.1.3 Conclusion 65
6.2 Q2 — Productivity 66
6.2.1 Analysis of Compilations 66
6.2.1.1 Comparison of Proportions 66
6.2.1.2 Comparison of Averages 67
6.2.2 Analysis of Time 67
6.2.2.1 Total Time Per Response Category 67
6.2.2.2 Time Per Compilation For Each Response Category 68
6.2.3 Conclusion 68
6.3 Q3 — Phases of Compilation 68
6.3.1 Assignments — Analysis of Compilations 69
6.3.1.1 Comparison of Proportions 69

6.3.1.2 Comparison of Averages 70
6.3.2 Laboratory Quiz 70
6.3.2.1 Analysis of Compilations 70
6.3.2.2 Analysis of Time 71
6.3.3 Conclusion 72
6.4 Q4 — Frustration When Fixing Errors 72
6.4.1 Conclusion 72
6.5 Q5 — Confidence in Programming Ability 73
6.5.1 Confidence Change Over Time 73
6.5.2 Comparison of Control And Enhanced Groups 73
6.5.3 Conclusion 74
6.6 Q6 — Compiler Appreciation 74
6.6.1 Conclusion 74
6.7 Q7 — Self-assessed Versus Measured PPE 75
6.7.1 Conclusion 75
6.8 Q8 — Participant Performance 75
6.8.1 Conclusion 76
6.9 Top Ten Most Common Errors 76
6.9.1 Comparison of Assignments and Laboratory Quiz 76
6.9.2 Comparison of Assignments and Similar Research 77
6.9.2.1 McCall and Kölling 77
6.9.2.2 Becker 78
6.9.2.3 Rountree 78
6.9.2.4 Jadud 78
6.9.2.5 Jackson *et al.* 78
6.9.3 Conclusion On Error Types 79

7 Conclusions **80**
7.1 Summary of Conducted Study 80
7.2 Conclusions and Recommendations 81

Bibliography **86**

A Tables and Forms **89**
A.1 Tables 89
A.2 Diagrams 102
A.3 Information Letter 104
A.4 Withdraw Form 108
A.5 Programming Experience Pre-assessment 110
A.6 Questionnaires 114
A.6.1 Questionnaire 1 114
A.6.2 Questionnaire 2 116
A.6.3 Questionnaire 3 118
A.7 Laboratory Quiz 120
A.8 Enhanced Error Messages Used In Study 131

LIST OF FIGURES

1.1 A Typical Novice Programmer's First Java Program 2

A.1 Assignments — Top Ten Most Common Errors 103

A.2 Laboratory Quiz — Top Ten Most Common Errors 103

LIST OF TABLES

A.1 Assignments — Total Number Of Errors Encountered Per Compilation Phase. Okay = No Errors Detected 89

A.2 Assignments — Total Number Of Errors Encountered Per Compilation Phase — χ^2-test Results. Okay = No Errors Detected 90

A.3 Assignments — Average Number Of Errors Encountered Per Participant For Each Compilation Phase. Okay = No Errors Detected . 90

A.4 Assignments — Average Number Of Errors Encountered Per Participant For Each Compilation Phase — t-test Results. Okay = No Errors Detected 90

A.5 Laboratory Quiz — Total Number Of Errors Encountered Per Compilation Phase. Okay = No Errors Detected 91

A.6 Laboratory Quiz — Total Number Of Errors Encountered Per Compilation Phase — χ^2-test Results. Okay = No Errors Detected 91

A.7 Laboratory Quiz — Average Number Of Errors Encountered For Each Compilation Phase. Okay = No Errors Detected 91

A.8 Laboratory Quiz — Average Number Of Errors Encountered For Each Compilation Phase — t-test Results. Okay = No Errors Detected 92

A.9 Laboratory Quiz — Distribution of Participant Responses to Error Messages 92

A.10 Laboratory Quiz — Distribution of Participant Responses to Error Messages — χ^2-test Results 93

A.11 Laboratory Quiz — Average Number of Compilations Per Participant For Each Response Category 93

A.12 Laboratory Quiz — Average Number of Compilations Per Participant For Each Response Category — t-test Results 94

A.13 Laboratory Quiz — Total Time Spent (in seconds) On Each Compilation Phase. Okay = No Errors Detected 94

A.14 Laboratory Quiz — Total Time Spent (in seconds) On Each Compilation Phase — t-test Results. Okay = No Errors Detected 94

A.15 Laboratory Quiz — Average Time Spent (in seconds) Per Compilation For Each Compilation Phase. Okay = No Errors Detected . . . 95

A.16 Laboratory Quiz — Average Time Spent (in seconds) Per Compilation For Each Compilation Phase — t-test Results. Okay = No Errors Detected 95

A.17 Laboratory Quiz — Total Time Spent (in seconds) On Each Response Category . . . 95
A.18 Laboratory Quiz — Total Time Spent (in seconds) On Each Response Category — t-test Results . . . 96
A.19 Laboratory Quiz — Average Time Spent (in seconds) Per Compilation For Each Response Category . . . 96
A.20 Laboratory Quiz — Average Time Spent (in seconds) Per Compilation For Each Response Category — t-test Results . . . 97
A.21 Laboratory Quiz — Participant Performance On Laboratory Quiz. P = Perfect, I = Imperfect, N = Not Attempted . . . 97
A.22 Laboratory Quiz — Participant Performance On Laboratory Quiz — χ^2-test Results. P = Perfect, I = Imperfect . . . 98
A.23 Laboratory Quiz — Number Of Compilations For Each Laboratory Quiz Question. P = Perfect, I = Imperfect . . . 98
A.24 Laboratory Quiz — Number Of Compilations For Each Laboratory Quiz Question — t-test Results. P = Perfect, I = Imperfect . . . 99
A.25 Laboratory Quiz — Time Spent (in seconds) On Each Laboratory Quiz Question. P = Perfect, I = Imperfect . . . 99
A.26 Laboratory Quiz — Total Time Spent (in seconds) On Each Laboratory Quiz Question — t-test Results. P = Perfect, I = Imperfect . . . 100
A.27 Questionnaires and Programming Pre-assessment — Descriptive Statistics. PPE = Prior Programming Experience . . . 100
A.28 Questionnaires and Programming Pre-assessment — t-test Results. PPE = Prior Programming Experience . . . 101
A.29 Change in Confidence Over Time Within Each Group — Paired Sample t-test Results . . . 101
A.30 Comparison of `javac`'s and *`Decaf`*'s Error Messages . . . 142
A.31 *`Decaf`* Exclusive Error Messages . . . 144

Chapter 1

Introduction

This thesis was completed as part of my graduate studies at the University of Northern British Columbia (UNBC). UNBC is a small research university of about 3500 students located in Canada's western-most province of British Columbia [28]. A super-majority of UNBC's students are from British Columbia (myself included). UNBC offers a variety of programs including Computer Science (which was my major during my undergraduate studies). The participants for the study featured in this thesis were UNBC students attending CPSC110 (that is, Introduction to Programming For Non-Majors).

Many experts agree that programming is a difficult task to learn for a variety of reasons. One notable reason is that many compilers that are used commercially were designed for experts, not beginners. Another reason, and one that is related to the previous, is inadequate compiler error messages. Many error messages do not accurately inform the programmer about the actual cause of the error. Consider the Java program in Figure 1.1 on the following page.

This program contains one syntax error; namely, the opening brace at the end of line two is omitted. Compiling this program with `javac` does not report that a brace is missing but it instead reports "`; expected`" This is unfortunate as no number of semicolons will fix this problem. First year, and especially first semester,

```
1 public class HelloWorld {
2         public static void main(String[ ] args)
3                 System.out.println("Hello World!");
4         }
5 }
```

Figure 1.1: A Typical Novice Programmer's First Java Program

computer scientists often take error messages at face value as they do not know that the compiler can lead them astray. Furthermore, these programmers may not know how to consult online resources for advice on errors that are difficult to fix. This results not only in decreased productivity (as the programmer spends a lot of time to fix something that should take no more than a few seconds), but also unnecessary frustration.

I was unsatisfied with the state of compilers used by myself and my peers when learning how to program in our undergraduate Computer Science programs. In response, I dedicated this thesis to finding a way to help novice programmers be less frustrated and more productive not only when fixing errors but also when adding new features to programs. The first step in doing so was to educate myself on what programming errors are made by students and why. It was at this time that I discovered the concept of enhanced compilers. Enhanced compilers attempt to address the problem of poor error messages by providing further insight into causes, and potential solutions, of detected errors. Compiling the program in Figure 1.1 with the *Decaf* enhanced compiler would generate an error message that there is one fewer opening brace then closing braces and that this may be the cause of the error. This pushes the programmer to take a closer look at their braces where they are more likely to notice the absence of the opening brace at the end of line two. I believed that enhanced compilers could be effective at helping beginners make less errors, be more productive, and achieve more success but I also wanted to also wanted to be through and investigate other options as well. As a result,

I examined a portion of the literature that provides alternatives to compiler error messages. However, the solutions were not suitable for the first time programmers that I wanted to study so I settled on enhanced compilers as my approach to helping programming novices overcome poor error messages. Specifically, I choose `Decaf` as it was the most appealing of the enhanced compilers that were considered. I also made note of the non-compiler tools that I felt could be helpful in completing my thesis. My complete literature review, which contains the four topics discussed above, can be found in Chapter 2 on page 6.

The problem of loss of productivity due to poor compiler error messages is one that I feel should be addressed. Inadequate error messages are frustrating for learners of programming languages and previous work has shown that frustration can lead to lower academic performance and higher attrition rates. Getting stuck on simple errors because of unhelpful compiler error messages surely does not do any wonders for the confidence of novices. I believe that enhanced compilers have the potential to address all three of these problems. These questions are discussed in greater detail in Chapter 3 on page 27.

My thesis focuses on more than the three problems described above. Specifically, the enhanced compiler's effect on productivity while answering programming problems, the number of errors generated by students, the types of those errors, frustration experienced while fixing errors, confidence in programming ability, academic performance, and the appreciation of the compiler used are examined with appropriate statistical tests. The explicit research questions that are answered in this thesis are located in Section 4.15 on page 43.

Some components of my study, such as the focus on the enhanced compiler's effect on the number of errors generated, is similar to previous work conducted by other authors. In other cases, such as the effect of the compiler on productivity and frustration, there is little historical data to compare my results to. In either case,

it is useful to compare and contrast the aims of my thesis with other work. This literature comparison is located in Chapter 2 on page 6.

In order to make the most of this opportunity, it was necessary to carefully design an empirical study that would allow me to answer all of the research questions discussed above. I choose CPSC110 for this purpose as I believed this was the best way to control for my participants' prior programming experience. As a precaution, I also held a programming pre-assessment to ensure that my participants were novices. Two groups of approximately equal size were formed. The first group, named the control group, used the standard Java compiler `javac`. The second group, name the enhanced group, used the enhanced Java compiler *`Decaf`*. Over the course of the semester, the participants completed a number of assignments and questionnaires in addition to a single laboratory quiz. The laboratory quiz was treated similarly to an exam; no talking, no cheating, no leaving the room and so on. However, the participants were informed that the laboratory quiz was not graded and would not effect their final grades. After the final grades for the course were made available, it was necessary to determine if the enhanced compiler had affected the academic performance of its users. If a significant difference in final grades between the groups was found, the weaker group would be compensated. Preparing the database containing all of the snapshots of the participants' programs for statistical tests was the next step. The final component was conducting all of the necessary statistical tests that would address my research questions. For more information on the research methodology that was implemented this thesis, see Chapter 4 on page 31.

The statistical tests used to analyze my results were an important part of my thesis. The most common test I used was the independent samples t-test. This type of test is ideal for comparing the means of two groups on some item (such as the frustration experienced when fixing errors). Independent samples t-tests use

Cohen's d to indicate how different the means are between the two groups that are examined (also known as the effect size). Another common test I choose was the χ^2-test, which is a type of statistical test that is great for comparing a set of proportions between two groups. This was the test that was used to determine if one of the groups made significantly more or less errors for each of the phases of compilation. Cramer's V was the effect size used for the χ^2-tests. There was also one instance where I used a paired-samples t-test to compare that confidence in programming ability between the control group and the enhanced group at three different points in time. Lastly, Pearson's correlation was the test of choice to determine the relationship between a participant's self-assessed Prior Programming Experience versus their scores on the programming pre-assessment. The results of the statistical tests mentioned here can be found in Chapter 5 on page 57.

The results of my thesis are quite promising. The enhanced compiler *Decaf* was shown to significantly reduce the number of errors generated by students. Students that used the enhanced compiler were able make to make progress on their errors, especially semantic errors, faster than those who did not. Fixing errors proved to be significantly less frustrating for programmers who learned to program with *Decaf*. And lastly, the students who used *Decaf* underwent a greater increase in their confidence than students who used *javac*. For additional discussion on my interpretation of the test results, see Chapter 6 on page 64. My concluding remarks and recommendations that follow from these results can be found in Chapter 7 on page 80.

Chapter 2

Literature Review

In this section, I summarize previous research on enhanced compilers and other literature regarding student learning. There is much literature regarding compiler error messages, including error message structure, weaknesses, and alternatives. The literature regarding error message structure was an influencing factor on my choice of enhanced compiler. Error message weaknesses has been recognized as a problem for a long period of time; the first literature on this topic dates back to 1976 [27].

Before conducting my research, I knew that I wanted to assist programming novices with inadequate compilation error messages. In order to contribute on this matter, I needed to investigate what errors are made by students and why. This topic also included some research on the effectiveness of various error message structures as well as the responses made by students in response to error messages. It was at this point in my literature survey that I came across a potential solution to this problem in the form of enhanced compilers. This solution appealed to me so I collected a number of articles on enhanced compilers. However, I also wanted to investigate other alternatives to compiler error messages in the event that there was an even better solution to improving poor messages than enhanced compilers. I also came across some non-compiler tools that I thought would be helpful in the

data collections and analysis portions of my thesis.

The articles I found during my literature review naturally fell into one of the four topics described above. I felt that these categories would be a useful for keep related articles together. For the articles that discuss an alternative to compiler error messages, see Section 2.1. For the collection of work that investigates errors made by students and other information on error messages, see Section 2.2 on page 9. The enhanced compilers that I considered for use in my thesis are located in Section 2.3 on page 18. Lastly, the non-compilers tools that I thought had potential in completing my thesis are described in Section 2.4 on page 24.

2.1 Alternatives To Compiler Error Messages

Some researchers were unhappy with the state of compiler error messages. In some cases, these researchers attempted to solve this problem by creating alternatives to poor messages. The alternatives to compiler error messages have been catalogued here.

Brown, 1983 Brown is one researcher who sought an alternative to compilers. As an aside in [7], Brown offers support for menu-driven programming; where programmers use menus to create various programming constructs. The primary advantage of menu-driven programming being that it reduces or even entirely eliminates syntax errors and the need for a compiler as the constraints force the programmer to create syntactically correct programs. While the possibility of completely eliminating syntax errors appeals to me, I worry that students who learn to program using menu-driven programming will be unprepared if they are ever required to use more traditional styles of programming. As such, I feel that menu-driven programming is unsuitable for my thesis and will not be using it.

Barik *et al.*, 2014 Barik *et al.* [3] also investigated alternatives to compilers. Their alternative to compiler error messages was the creation of taxonomies for different kinds of errors and a prototype IDE that displayed this information for the programmer. Potential solutions to errors were suggested based on the relevant taxonomy. The code segments relevant to the error were highlighted for increased visibility. An advantage that this approach has is that the taxonomies and even the suggested solutions may be applied across multiple programming languages.

Barik *et al.* note a weakness of the prototype IDE; it is not well equipped to handle "bad practices" as the fixes to these issues tends to be more subjective than other fixes. A second weakness is that the effectiveness of the prototype IDE is highly dependent on the quality of the error messages provided by the compiler that is being used. This second weakness is important as the entire field of enhanced compilers is the direct result of the low quality error messages produced by many compilers.

The approach that I am taking with my thesis can be thought of as the opposite of what was done by Barik *et al.* Instead of creating new taxonomies, I will use the categories of errors that are present in `javac` which are then used by `Decaf` to provide enhanced error messages. Barik *et al.* also use a prototype IDE in their research. Creating an IDE is not a necessary, nor important, part of my thesis as `Decaf` includes an editor for Java programs.

Campbell *et al.*, 2014 Campbell *et al.* [8] believe that compiler error messages, especially in regards to syntax errors, can be frustrating. Campbell *et al.* address this issue with a tool for Java that creates an N-gram language model of source code tokens. This model predicts what a project should look like based off of previous, error-free versions of the project. The tool flags code segments that are not in line with the models predictions as suspicious and presents this information to the pro-

grammer. Programmers can then examine suspicious code segments to determine if there are any syntax or semantic errors that need to be fixed. The model was found to outperform the `javac` commercial compiler when a correct version of the project was available for referencing. The tool was also found to perform better on smaller projects.

While Campbell *et al.*'s work provides promising results, it is not suitable for my thesis. The assignments and laboratory quiz questions completed by my participants are relatively small in scope (solutions always consisted of one class and often just one method). Furthermore, there was seldom reason to change code segments that contained no errors as there was little reason to add more once the code segment was completed for the first time. As a result, the suspicious code segments flagged by the model for further review would likely always be the most recently added feature. This is something that `javac` could inform the programmer, which diminishes the usefulness of the model in the context of implementing it for my thesis.

2.2 Investigating Student Learning and Errors

Programmers have been making syntax and semantic errors in their programs for many years. Research into this topic appears to have become more rapid in the last decade. The topic of the errors made by students and the messages they receive from the compiler is an important component of my thesis and is where I found the most literature that could be used in my thesis. A description of each article I found, as well as some observations and its relvance to my research topic, is located below.

Brown, 1983 Brown examines the quality of error messages produced by a number of compilers for the Pascal programming language [7]. Each compiler was fed

a simple Pascal program that contained a single syntax error; a missing opening parenthesis when calling a function. Nearly all of the compilers produced error messages of poor quality; very few of the compilers were able to both identify the error correctly as well as the location of the error.

Brown's work confirms an observation I made prior to beginning my research: namely, that compiler error messages are often poor as they struggle to correctly identify the actual error made by novice programmers. I believe that enhanced compilers may be effective at addressing poor error messages and previous work in this field would agree.

Jackson *et al.*, 2005 After working on the enhanced pre-compiler for Java named `Gauntlet`, Jackson *et al.* interviewed faculty United States Military Academy on what they thought were the most common errors expected by students [17]. Jackson *et al.* created an automated error collection system to determine if the faculties' expectations were in line with reality. This system logged all of the syntax and semantic errors made by both students and faculty using an IDE over the course of a semester. Five of the top ten most frequent Java compilation errors encountered by students are not in line with what the faculty was expecting. The error "Illegal start of expression" was the third most common error that was encountered and yet it was not in the faculty's top ten list. It is noted that novice programmers may struggle to fix this error for two reasons. First, illegal start of expression is an error message that is generated by many different causes, meaning the solution that fixes the error is not immediately obvious. Second, the term "expression", as it pertains to Java, is unfamiliar for novice programmers. Conjecture about typical causes for some of the other most common errors is also presented by Jackson *et al.*

The observation that an error message can be generated by multiple causes is important for enhanced error messages to take into account if they want to be their

explanation for why an error occurred to be accurate and useful. One important difference between mine and Jackson *et al.*'s data sets is that I want to collect more than just the error messages generated by students. By collecting only the errors that were detected, there is not enough information to assess the effectiveness of a participant's response to that error message. This is why, in addition to collecting error messages, I am also collecting a snapshot of the participants' program at at the time of compilation. The snapshots will allow me to see everything that a programmer changed from one compilation to the next which in turn makes it much easier to determine the effectiveness of the programmer's response to the error message.

Jadud, 2005 and 2006 In 2005, Jadud investigated novice compilation behaviour and states that this particular field is not well understood [18]. His research focuses on two questions: what novice students compile and when they do the compilations. Java is the programming language used in Jadud's study. That data collected suggests that beginner programmers tend to write large amounts of code all at once; frequently working for at least five minutes before compiling. The students then make an effort to fix all of the errors that are present in their program. Jadud notes that novice programmer behave differently once they have made an error. In this scenario, half of the students' compilations occur with twenty seconds of the previous compilation. However, this fact does not necessarily indicate that the novice programmers have fixed errors quickly; the data shows that many of these compilations result in a program that still contains syntax or semantic errors. Lastly, Jadud examined which types of errors were the most common. He found that the three most common errors were trivial syntax errors that could be fixed quickly and with few keystrokes by the novice programmers.

Jadud continued to investigates methods and tools for exploring novice com-

pilation behaviour [19] the year after his previous work on this matter. Included in his research is an examination of the "edit-compile cycle", which refers to the the behaviours students engage in as they edit and compile their programs. CS1 students were observed learning how to program in Java over the course of two years. Students are introduced to programming in an objects first approach with the use of the `BlueJ` IDE. A snapshot of a student's program was recorded every time the program was compiled in the computer laboratory at the University of Kent. Other recorded data types include: the types and frequency of errors and time elapsed between compilations. The paper also includes a small case study in the form of an analysis of a novice programmer and his attempts at a fixing programming errors with the help of a compiler.

Jadud suggests that the final version of a student's program is not indicative of how much they struggled to get to that point. Instead, Jadud presents a measure called "Error Quotient" (abbreviated as EQ). EQ is a measure of how well a particular programming session played out for a student. EQ is normalized between 0 and 1. An EQ of 0 means no error persisted through two consecutive compilations, a score of 1 means that the same type of error was present in every single compilation. As a student's score approaches 1, the instructor's belief that the student is struggling with debugging becomes more substantiated. An algorithm is given for calculating a student's EQ, as well as details on how the algorithm can be modified to give a different distribution of EQ scores. Jadud also studied the relationship between a student's EQ and the grade that student achieved on the written final exam and found it to be weak.

It can be argued that Jadud and I are both examining the behaviour of novice programmers. Specifically, productivity is the aspect of behaviour that I am examining. There are similarities in the types of data collected over the course of our studies such as the types and frequencies of errors generated as well the time

between compilations. While Jadud's theory on measuring programmer struggles with Error Quotient is interesting, I will not be using it in my thesis as I feel there my needs on this front have been satisfied with the rubric produced by Marceau *et al.* [22].

Nienaltowski *et al.*, 2008 Nienaltowski *et al.* study various compiler error message formats to determine if there are any particular formats that are remarkable; for good or bad reasons [26]. They note that there are three main approaches for designing compilers that are helpful for beginners. The first is to build new compilers from scratch. The second is to improve existing compilers by modifying them. The third is to improve only the error messages reported without modifying the compiler's algorithms.

The format of error messages belongs to one of three categories; short form, long form, and visual form. The short form of error message is often used by production compilers such as `javac`. Short form error messages include the location where the error occurs, the code segment containing the error, the type of error, and a description of the error. Long form error messages include everything a short form message has and more. Most notably, long form messages include the token that resulted in the the error and possibly a suggestion for fixing the error. Visual form error messages are often part of IDEs such as `BlueJ`. The erroneous code is highlighted (or is otherwise marked in a visible fashion) and a brief description of the error is provided. I would argued that the error messages produced by enhanced compilers such as `Decaf` fall into the long form category.

The data collected by Nienaltowski *et al.* shows that students with higher levels of experience were able to recognize errors in written code segments more quickly than students with less experience. Additionally, the experienced students were more likely to provide the correct fixes for the errors. The observations hold re-

gardless of the format of the error message. Visual form error messages were found, relative to short form messages, to improve response times to errors at a cost of decreasing the ability to correctly fix the errors for students of all experience levels with beginners seeing a greater decrease. Long form messages were not found to improve a student's ability to fix an error correctly nor were they found to improve a student's response time relative to short form error messages.

Of the three strategies for designing compilers that are useful for beginners that are discussed by Nienaltowski *et al.*, `Decaf` definitely belongs to the third category as, for the most part, as it receives the error messages from an unmodified `javac` and rewords them in layman's terms for ease of understanding. There is overlap in between my thesis and Nienaltowski *et al.*'s study in that we both examine if programmers with enhanced error messages (i.e. long form messages) are able to correct error with better success and in less time than programmers who use short form messages. One difference is that partial fixes for errors are treated positively in my thesis and negatively in Nienaltowski *et al.*'s work.

Rountree *et al.*, 2009 `ClockIt`, the data collection tool created by Rountree *et al.*, was used in the examination of the development practices of introductory computer science (CS1) students [12]. CS1 students participated in a study that took place throughout Fall 2007 and Spring 2008. The data collected on novice development practices was then compared against the results of a similar study performed by Jadud [19]. Lastly, Rountree *et al.* note that the possibilities of making higher level observations from the data collected, such as which students may be cheating and the effect of starting assignment later than recommended. Several different kinds of data regarding the CS1 student's programming habits are collected by Rountree *et al.* For example, the types and frequencies of various compiler errors is recorded as well as the amount of time between compilations of programs. Addi-

tionally, the amount of time that students of varying performance levels required to finish their programming assignments is also recorded. One observation that can be made is that students that achieve an F grade spend much more time on assignments that their C grade peers, and nearly as much as B students. A possible explanation for this is that F students put effort towards their assignment but they may not be able to fix all of the errors that they encounter (as they would be able to achieve more than an F if they did).

There are some similarities between my work and Rountree's as we both investigate the types and frequencies of errors generated by participants as well as the time that elapses between compilations. Consequently, there are also similarities between my work Jadud's [19] which were discussed previously in this section. `ClockIt` was one of the tools I considered for my thesis at it may have made collecting the snapshots of the participants programs as well as measure student productivity easier. However, I was informed that `ClockIt` had not been in recent development and would likely be incompatible with newer versions of `BlueJ`. The daunting time lines of my thesis proposal did not give me confidence that I could update the software and test it for correctness before it was needed for CPSC110.

Barik *et al.*, 2014 Barik *et al.* examine the construction of compiler error messages [2]. They observe that programmers can understand what the error message is trying to convey if they "think-aloud" during the process. Various forms of error message annotations were tested. The purpose of the annotations is to reveal the compiler's internal reasoning for why a particular code segment was erroneous. Visual annotations were found to be the most effective for this purpose. Barik *et al.* noted that programmers began incorporating visual annotations into their own explanations after encountering them for the first time.

Barik's results are interesting though I am curious if the effectiveness of visual

annotations changes with the amount of experience held by the programmer. I wanted to avoid visual form error messages and annotations for the enhanced error messages in my thesis. This is because I believed it would be more meaningful to draw a comparison between the text-based `javac` and text-based enhanced error messages.

Kölling *et al.*, 2014 Kölling *et al.* examine the difference between "logical errors" and "compiler errors" [23]. They note that many previous works in this field focus on which types of errors are the most common. Errors are categorized based off of compiler error messages in these works. One example of a compiler error message category is "`; expected`". Kölling *et al.* instead chose to categorize errors based on logical errors. The logical error reflects the actual mistake that a programmer has made and this can differ from the mistake that is detected by the compiler. Kölling *et al.* make two important observations that reveals a flaw with using only compiler error message types to categorize errors; First, a given type of logical error can produce different compiler error messages depending on the context in which the logical error occurs. Second, the opposite was also found to be true; different logical errors can produce the same error message. It is worth noting that Jackson *et al.* also noted that different errors can produce the same error message [17]. Kölling *et al.* conclude by advising researchers to be careful when dealing only with relative frequencies of compiler errors as this may not tell the entire story.

I agree with Kölling *et al.*'s warning that researchers should be careful with relying on the relative frequencies of compiler errors. I considered categorizing errors by logical errors rather than by compiler errors for my thesis. However, I was unable to create a consistent scheme for classifying logical errors in time for CPSC110. My hope is that future work in the area of enhanced compilers will address this issue by using logical errors for their error classifications.

Guzdial, 2015 Guzdial investigates the practice of teaching program to novice computer scientists by having the students practice on their own, also known as "minimally guided instruction" [15]. Guzidal was inspired by an earlier study that compared two groups of math students one which were given worked out problems while the other group had to solve them on their own. After the first set of problems, each group then had to solve a second set of problems. The group that was given the pre-worked problems solved the second set of problems faster and with fewer errors than the other group. This finding is extended to computer science; programmers have trouble writing programs when they are still learning how to read and understand the programming language.

I believe enhanced compilers may be effective at helping programmers learn how to program by providing a level of guidance above "minimally guided." My thesis will evaluate if this theory or valid or not by measuring the productivity of novice programmers, the effectiveness of their responses to error messages, and their academic performance.

Munson and Schilling, 2016 Munson and Schilling analyze novice programmers' response to error messages [25]. They discussion how the cycle of "`edit` → `compile` → `interpert errors`" is used by some first time learners to learn the syntax of a programming language. A criticism they have of compilers is that compilers report "program-translation problems", or the problems encountered as the compiler tries to turn the source code into executable code. This is problematic as the program-translation problem may not reflect the actual error the student made (the similarities between this particular finding and the findings by Jackson *et al.* [17] and Kölling *et al.* [23] should be noted). As an example, experienced students recognize that a "`; expected`" error message does not necessarily mean that a semicolon is missing in their code. Beginners were observed to struggle with

learning this concept. Furthermore, beginners struggle to learn the concept that compiler errors beyond the first can be the result of the compiler being confused by the first error and are not necessarily worth addressing. Students that achieved higher grades were found to be more likely to fix the first reported error first, but the opposite does not hold. Lastly students that achieved higher grades spent more time in the edit phase of the cycle discussed above.

I believe that enhanced compilers may be effective at addressing the problem of "an error of '; missing' does not necessarily mean a semicolon is missing" as enhanced compilers can provide suggestions about multiple potential causes of the error that the programmer should investigate. I am not surprised that students that addressed the first detected error first outperformed their peers as I view this as a good practice. One of the reasons the *Decaf* enhanced compiler appealed to me was that it follows this philosophy by only showing the first error that is detected. I am also not surprised that students that spend less time fixing errors also achieve more academic successful than students who spend a lot of time fixing errors. As a consequence of this, I feel that enhanced compilers may help improve the grades of its users if the compiler allows the novice programmers to fix errors faster.

2.3 Enhanced Compilers

When designing my study, it was necessary to chose the enhanced compiler with care as it was a crucial component of my thesis. Complicating matters were the two requirements set by the instructor of CPSC110 that the enhanced compiler had to meet. An enhanced compiler that failed either requirement was thus unsuitable for my thesis. The first requirement was that the enhanced compiler was for the Java programming language as this was the basics of this language were going

to be taught to the class. The second requirement was that the enhanced compiler had to be compatible with an Integrated Development Environment (IDE). An IDE can be thought of as an editor that can be used to create programs and they often contain features that make this task simpler for the programmer. One such feature is that IDEs often allow programmers to compile their programs without having to use a command-line shell. It could be possible to take an existing IDE and modify its inner workings to use an enhanced compiler. However, the aggressive time lines that were present during my thesis proposal left no time for testing if the resulting combination of software worked correctly and was free of bugs. These two requirements had the effect of considerably narrowed the number of enhanced compilers that were suitable for my thesis.

Lewis and Mulley, 1998 Lewis and Mulley [21] believe that commercial compilers are unsuitable for learning a specific programming language and for learning how to program in general. An enhanced compiler for the Modula-2 programming language was tested with novice and experienced students. This is in contrast to my thesis which explicitly focuses only on novice students with as little prior programming experience as possible. The relative frequencies of syntax and semantic errors was recorded. The data shows that beginning students mostly make syntax errors while experienced students mostly make semantic errors. It is also suggested that the enhanced error messages have a larger impact on novices with regards to reducing the overall frequency of errors.

Lewis and Mulley's enhanced compiler was considered but proved to be incompatible with my thesis. It is true that the second requirement described earlier was satisfied as Lewis and Mulley's enhanced compiler was designed to be used with the `Ceilidh` IDE. However, the enhanced compiler was for the Modula-2 programming language and not Java. This is the primary reason why the enhanced

Modula-2 compiler was not chosen for my thesis.

Hristova *et al.*, 2003 Hristova *et al.* [16] observe that many textbooks cover basic compiler errors in some fashion, but this is not sufficient to prevent students from making the errors. Faculty members were interviewed about what they thought the five most common types of errors were. This data was used to create an enhanced pre-compiler for the Java programming language named *`Expresso`*. That is, *`Expresso`* is used to check a student's programs for common mistakes before using *`javac`* to check for all possible syntax and semantic rule violations. An evaluation of *`Expresso`*'s effectiveness was left for future work. *`Expresso`* was inspired by other tools such as *`TA Online`*, *`DrScheme`*, and *`BlueJ`*. *`TA Online`* is a resource that lists many common Java errors that students tend to make. However, *`TA Online`* is intended to be used as a reference as it goes not interact with the programs written by students. The lack of interaction made *`TA Online`* unsuitable for the needs of Hristova *et al.* *`DrScheme`* is interactive but is for intended for the Scheme programming language rather than Java. Hristova *et al.* note that Expresso addresses many of the same problems that are addressed by *`DrScheme`*. The examination of *`BlueJ`* suggests that it can cause beginner programmers to have poor understanding of some programming concepts, possibly due to *`BlueJ`*'s large amounts of code auto-completion. For example, *`BlueJ`* automatically creates the main method of a program when starting a new project; novices might not realize that they need a main method, or they might not know how to properly create a main method, since they do not write it by hand.

I previously mentioned that there were two requirements that the enhanced compiler had to satisfy to be considered for my thesis. I considered using *`Expresso`* for my thesis but it did not meet the second requirement of being compatible with an IDE. As such, *`Expresso`* was unsuitable for use in my thesis.

Jackson *et al.*, 2004 Jackson *et al.* [13] interview course instructors and found that student frustrations regarding compiler error messages were a concern for many of these instructors. Jackson *et al.* address the issue through with a pre-compiler called `Gauntlet`. One of the motivations for `Gauntlet` is that students were believed to be focusing too much on the fine details of the syntax of Java when this was not the purpose of the exercises. Furthermore, there were discrepancies between the errors that the faculty were expecting students to encounter versus the errors that were actually encountered. As a result, the faculty was unable to adequately prepare students for the challenges they would face.

`Gauntlet` is intended to make the act of programming more enjoyable for beginners and this is accomplished with more informative error messages (such as typical causes of the error), using humour, and praising the students for writing an error-free program. `Gauntlet` was originally designed for the Java programming language but it can be adapted to different programming languages. Jackson *et al.* evaluated the effectiveness of `Gauntlet` and observed that students that used `Gauntlet` submitted work of higher quality. All of the instructors who used `Gauntlet` in their classrooms stated that they believe that the system is a success. The instructors also noted a lessened workload as office hours were less tied up with syntax errors and assignments could be marked faster.

`Gauntlet` was one of the options for an enhanced compiler that was considered for my thesis. However, I was unable to reach the authors of `Gauntlet` and as such I was unable to use this enhanced compiler for my thesis.

Marceau *et al.*, 2011 Marceau *et al.* [22] note that the `DrRacket` programming environment for the Racket programming language makes a significant effort towards having helpful error messages; this is done through various sub-languages that correspond to the concepts that a student has learned. That is, a student

writing their first program may receive a different customized error message than someone with more experience and knowledge of the programming language. The effectiveness of `DrRacket`'s strategy to enhancing error messages was examined.

Snapshots of students' programs was collected each time the programs were executed, as well as the keystrokes done in between runs. To measure the effectiveness of an error message, a rubric was created that interviewers used to determine whether or not a students' response to the error was reasonable. Marceau *et al.* made two observations. First, syntax errors that occur early in the course are difficult for students to fix, possibly due to inadequate code highlighting. Second, syntax errors that occur later in the course were found to depend on the concepts that students were working with during the assignments. As such, looking only at the relative frequencies of errors within an assignment is a flawed approach. It is noted that many invalid expressions can be flagged by the compiler in multiple ways; closer inspection by a human was required to overcome this particular issue.

While `DrRacket` satisfies the second of the two requirements for my choice of enhanced compiler, it fails the first condition as it is not an enhanced compiler for the Java programming language. As such, I was unable to use `DrRacket` for my thesis. The collection of snapshots of students' programs did appeal to me and this is ultimately what I did in my thesis (one difference is that I collected for every compilation rather than every execution).

Becker, 2015 and 2016 Becker notes that programmers often encounter poor error messages early on in their careers. It is not uncommon for these poor error messages to manifest in the first program that is created by a novice programmer. Becker views this as problematic due to the fact that novice programmers have little to no experience with fixing errors. As such novice programmers must rely on the inadequate compiler error messages to help them. Becker attempted to address

this by creating an enhanced compiler for the Java programming language by the name of `Decaf`, which was originally presented in 2015 [4]. `Decaf` displays the error message from the `javac` commercial compiler alongside an enhanced error message. This may allow programmers to have a better understanding of how to correctly interpret the error messages from `javac`.

`Decaf` has been previously used in two studies. In the first study [5], Becker tests `Decaf` in a classroom setting with novice programmers. The data collected by Becker can be used to form several observations. The first is that programmers who learn how to program with `Decaf` make less errors overall than programmers who learn with the standard `javac` compiler. Second, the data shows that when `Decaf` users do encounter an error, they are more likely to fix the error within one compilation. In other words, `Decaf` users have less repeated errors. The version of `Decaf` used in this study only provided enhanced messages for thirty different types of errors; if the user makes an error that does not fall within these thirty, only the standard error message from the javac displayed. The data shows that `Decaf` users do not perform noticeably better or worse than their peers for the errors not enhanced by `Decaf` which suggests that it is the enhanced error messages that are helping students make less errors and not another factor.

The second study that `Decaf` was used in was focused on categorizing the errors made by novice programmers [6]. The categorization is done with the help of "Principal Component Analysis" (PCA). Becker *et al.* comment high dropout rates in introductory computer science courses can be at least partially attributed to difficulties with programming including difficulties that arise due to poor compiler error messages. Becker *et al.* note that studies often fall into one of two categories; the first category attempts to categorize errors made by students in the source code, the second categorizes compiler error messages (It should be noted that my thesis encompasses both categories). Becker *et al.* use PCA to find hidden relationships

between various compiler error messages. That is, if a programmer makes an error that generates a particular error message, PCA can identify other errors messages that are likely to be generated by the programmer. It was found that programmers who often forget closing braces are more likely to see, relative to programmers who do not make this error, error messages such as "else without if" and "reached end of file while parsing" as both of these error messages can be generated by the same actual error (missing closing braces). Other groups of errors were also determined. Becker *et al.* conclude that further investigation should be done on the groups of errors that have been found under the belief that, if the groups are successfully validated, intervention strategies specific to an error group can be devised.

I determined that using PCA to find hidden relationships among the errors generated by my participants was outside of the scope that I wanted to research. I did, however, consider using the enhanced compiler `Decaf` for use in my thesis. It meets both of the requirements set by the instructor for CPSC110 as it is an enhanced compiler for Java and it includes and IDE for editing Java programs. I also agree with `Decaf`'s philosophy of showing only the first error that was detected not only because this is a habit that I try to teach CS1 students (as every error detected past the first may or may not actually exist) but also because it would allow me to categorize student responses to errors more consistently as participants did not have the option of choosing which error to respond to. As such, I requested and was granted permission to use `Decaf` in my study.

2.4 Non-Compiler Tools

In order to speed the completion of my thesis, I wanted to use tools created by others so that I could dedicated more of my efforts in other places of my research. The non-compiler tools that I considered for use in my thesis are described below.

Kölling *et al.*, 2003 Kölling *et al.* worked on an IDE intended for beginning programmer students named `BlueJ` [20]. `BlueJ` is designed to make it easier for programmers to learn Object-Oriented Programming (OOP). Kölling *et al.* criticize other IDEs on three main points; lack of object orientation with the editor itself, overwhelming complexity (for beginners), and too strong of a focus on Graphical User Interfaces builders.

I considered using `BlueJ` but determined it was not suitable for my thesis. `BlueJ` appealed to me because it is compatible with extensions such as `ClockIt`. Unfortunately, `BlueJ` does not come with an enhanced compiler, though it may be possible to create an extension or change the inner workings of this IDE to include one. However, the time lines I faced during the proposal of my thesis made me feel doubtful that I could implement an enhanced compiler in this way and thoroughly test that it worked correctly with `BlueJ` before the beginning of CPSC110. One aspect of `Decaf` which appealed to me is that it was designed to be a combination of an IDE and an enhanced compiler. This combination allowed me to begin testing its feasibility for my thesis earlier than would otherwise be possible.

Rountree *et al.*, 2008 Rountree *et al.* present a tool called `ClockIt` [27, 30]. `ClockIt` logs the development practices of novices so that educators can learn which practices are effective and which are not. `ClockIt` consists of two components: an extension for the `BlueJ` IDE that acts as a data logger and visualizer, and a Web Interface. It is noted by Rountree *et al.* that there are publications dating back to 1976 regarding programming languages being designed in such a way that they are difficult to learn. `ClockIt` can used to find useful information such as the compilation success rates of students, the types of errors encountered, and the amount of time students work on their programs and when the work takes place. With the data that can be collected, educators can discover which students are cheating.

I discussed previously that I considered using `ClockIt`, but was informed that it was not in active development and would require updating to be compatible the new versions of `BlueJ`. As such, it proved to be unsuitable for my thesis.

Marceau *et al.*, 2011 Marceau *et al.* created a rubric that can be used to determine the effectiveness of error messages [22]. The rubric allows "interviews" to determine if a students' response to an error message (in the form of edits) is reasonable. An effort was made to ensure that the rubric could be applied consistently across different interviewers. Marceau *et al.* note that even subjective decisions were made with surprising consistency. The final version of the rubric places a student's response into one of five categories: wholesale deletion of all of the erroneous code, response unrelated to the error, response unrelated to the error but fixes another error, response related to the error but does not fix it, and fixes the error.

I mostly agreed with the categories in the rubric created by Marceau *et al.* This rubric was useful for my thesis as I needed inspiration for how to consistently categorize responses to error messages across not only individual snapshots but also across participant groups. However, as discussed in Section 4.15.2 on page 44, I felt it was necessary to modify the rubric with the addition of two more categories so that it would be ideal for my thesis and the data set that I had to work with.

Chapter 3

Problem Statement

I was often asked by my peers for assistance in troubleshooting errors in their Java programs throughout my undergraduate education. One observation I made was that my peers often experienced tunnel vision where their focus on the compiler's error message blinded them to other details that were critical to fixing their error. I continued to witness this sort of behaviour when I became a teaching assistant for CPSC 101 at UNBC. As a result of these experiences, I agree with the experts with regards to compiler error messages being inadequate and unsuitable for beginners. My hope is that, by providing more information and by pushing students into thinking of possible causes of an error, enhanced compilers will allow novices to experience less tunnel vision when debugging.

3.1 What Problem Needs Solving?

Students can encounter significant frustrations during debugging as a result of poor compiler error messages. The time students spend on fixing errors can be considered to be unproductive in the same way that a math student that tries to solve a problem via trial-and-error is unproductive; the student does not make progress on their assignment unless they stumble across the correct solution (which they

may not be able to replicate in the future!). Students should be equipped with software that allows them to work productively and achieve success; commercial compilers, at least for novice programmers, are not good enough.

3.2 Why Is This Worth Investigating?

An investigation into increasing student productivity is worthwhile as it would have three notable outcomes. First, students may be able to work more productively by using less time and effort to fix programming errors. Increased student productivity may allow students to finish their assignments faster. If achieved, this will lead to two further outcomes; either the students will have more time to relax and thus less stress (and potentially reduced attrition rates), or instructors can fit more concepts into their assignments (which will better prepare students for the future).

Second, students may experience less frustration while debugging as it will not be as difficult a task and they will have more support from the compiler. Frustration is a two-edged sword in learning. On one edge, there is evidence that frustration pushes subjects to be more creative and improves concept learning [32]. On the other side, too much failure-induced frustration can induce helplessness [32]. Furthermore, frustration is a reaction to stress [24] and previous work has shown that stress has a negative impact on university student graduation rates [29] and academic performance [34].

Lastly, students may have more confidence in their programming ability if they are able to do more without asking help. This may prompt students into experimenting more when programming and this can be a useful learning opportunity.

There is a reasonable body of literature on enhanced compilers. However, there has been little work in the area of their effect on student productivity and of their

effect on student frustration. I intend to lay the foundation for future work with my thesis.

Students in CS1 courses begin writing programs early on in their university careers (at UNBC, students write a "Hello World" program similar to 1.1 on page 2 within the first three weeks of classes). Novice programmers often struggle as they are expected to write correct programs in a language they have never used before while they are still learning the core concepts of the language. For example, the Java concepts of statements and expressions may not be taught in CS1 and as such students may not understand what *javac* is referring to with error messages like "`illegal start of expression`" and "`not a statement`". First year computer science courses often have laboratory assignments that feature programming questions. If students are not able to finish an assignment entirely within a lab session, they must work on their own with little support. This is done under the guise of learning to do something by doing it yourself, also known as "minimally guided instruction." It has been argued that novice computer scientists do not learn how to program well under minimally guided instruction [15]. If enhanced compilers successfully improve student productivity, then they can be used to help students write programs and provide guidance above minimally guided instruction.

3.3 Definition of Productivity

Webster Dictionaries defines productivity as "the power of producing, the state or quality of being productive" [31]. Productivity can be thought of as the ratio of output (object created) to input (resources consumed) with larger ratios representing higher levels of productivity.

In the context of my thesis, there are two types of output. The first is a completed programming problem. An example of a completed programming prob-

lem would be one of the questions on the laboratory quiz (see Appendix A.7 on page 120) or one of the questions on the students' weekly programming assignments. The second type of output would be a successful fix for an error in that is in the students' programs.

Regardless of which type of output is being considered, there are two input factors; time and effort. It is trivial to calculate how much time has elapsed between compilations since each compilation is timestamped (see Section 4.2 on page 34 for more information). Consequently, I am also able to calculate how much time was required to finish a programming problem or to fix an error. The second input factor, effort, is harder to quantify. I will be using the number of compilations required to achieve the output as a proxy for effort. For example, a student who can fix a `';' expected` error in three compilations uses less effort than a student who requires seven compilations to fix the same error.

Chapter 4

Research Methodology

The methodology described in this section is what was approved by both my supervisory committee and the Research Ethics Board (REB) as well as being the methodology that was ultimately carried out. The methodology that I originally proposed is compared to the methodology that was actually done on an item by item basis.

Section 4.1 on the following page discusses how the participants for the study were selected and placed into two groups. Also included in this section is how the participants were informed of and consented to the study. Section 4.2 on page 34 outlines the software that the participants used extensively over the course of the semester, which was installed on the laboratory computers. The programming pre-assessment, anonymous identifiers, information letter and consent form, and the withdrawal form were all distributed at the same time. The former two are discussed in Section 4.3 on page 35 while the latter two are discussed in Section 4.4 on page 37 and Section 4.5 on page 37, respectively.

The weekly assignments completed by the participants as a part of CPSC110 are discussed in Section 4.6 on page 38. Questionnaires 1 and 2 are discussed in Section 4.7 on page 38 and Section 4.8 on page 39 respectively. The laboratory quiz, where much of my data was collected, is described in Section 4.9 on page 40.

Questionnaire 3, the final questionnaire of my study, is detailed in Section 4.10 on page 40.

Section 4.11 on page 41 discusses what was done to correct mistakes in the participants' anonymous identifiers on the three questionnaires. Section 4.12 on page 41 examines the topic of how I determined if the enhanced compiler affected the grades of my participants and what response was required. Some of the operations that were done to prepare the mySQL database for use in statistical analysis is discussed in Section 4.13 on page 42. A description of how I examined the snapshots of participants' programs is provided in Section 4.14 on page 43 Lastly, Section 4.15 on page 43 discusses the research questions for the study, which components of the collected data are relevant, and which statistical tests were chosen to answer the research questions.

4.1 Participant Selection and Grouping

In typical CS1 classes there may be some variety in the amount of Prior Programming Experience that the students have coming in to the class. This is because CS1 is a required class for Computer Science majors (at least at the University of Northern British Columbia). As such, CS1 will include students who have never learned anything related to programming as well as students who have at least some experience writing programs. When designing my study, I was concerned that my results could be skewed if my two groups featured unequal amounts of students with Prior Programming Experience. I attempted to control for this by selecting CPSC110 as the class in which to conduct my study. CPSC110, or Introduction to Programming for Non-Majors, is a class intended for students who need a Computer Science course to graduate but do not want to declare a major in this field. I argue that the students in CPSC110 are less likely to have any Prior Programming

Experience as students that are interested enough in programming to have Prior Programming Experience are much more likely to major in Computer Science and take the course intended for majors instead. In this regard, my study differs from others conducted in the field on enhanced compilers as the other studies often take in place in CS1 or CS1-like environments where a variety of Prior Programming Experience can be reasonably expected [21] [13] [22] [5].

At the beginning of the semester, CPSC110 students were informed of the study and were given the choice of opting out. It was made clear that students could refuse consent without penalty and that they could withdraw at any time also without penalty using a Withdraw Form (see Appendix A.3 on page 104 for the consent form and Appendix A.4 on page 108 for the withdraw form that were used). Participants also had the option to opt-out by ticking the appropriate boxes on each of the three questionnaires (see Appendix A.6 on page 114). By the end of the study, three participants had withdrew; all via the questionnaires.

The class of approximately fifty students was split into five lab sections. Each lab section was deemed to be either part of the "control group" or the "enhanced group"; this was done to preserve the independence of the two groups and to avoid situations where participants in the control group could easily that some of their peers had enhanced error messages while they did not. The lab sections were chosen such that the control group and the enhanced group were of approximately equal size. After accounting for the participants who withdrew, the control group contained twenty-three participants while the enhanced group had a size of twenty-five. However, these numbers include participants who dropped the class but did not choose to withdraw from the study. These participants still contributed to the data used in my statistical tests.

The enhanced compiler `Decaf` [5] is bundled with an editor for creating Java programs. This editor has an option to enable or disable enhanced error messages.

The participants in the enhanced group had `Decaf` configured such that enhanced error messages are enabled. For participants in the control group, enhanced error messages were disabled.

This component of my methodology saw large changes from between how it was original proposed and its final version, though the Withdrawal form and the process with which groups were created were present at all stages. The first draft of my participant selection originally involved students providing consent to the study when registering for the class. However, this approach was denied in part due to the finalization of my methodology occurring after most of the students had registered for the class. The next version of the methodology included an opt-in study and an incentive in the form bonus credit to the final grades for all participants of the study. In this version, potential participants would have been informed of the bonus credit (and the fact that withdrawing from the study would void the credit) before consent forms were distributed. The final version of the methodology switched to an opt-out study with no incentive. One of the reasons that drove the switch to an opt-out study was the maximize the sample sizes for my statistical tests as that would enable me to get the most meaningful results from this research opportunity.

4.2 Laboratory Computer Setup

Each laboratory computer had `Decaf` installed. Configuration files for `Decaf`, which included the toggle for enhanced error messages, were generated for each user based off of their username. This enabled a participant to have access to the same type of error message throughout the entire study, regardless of which lab computer was used and when. It also eliminated the possibility of participants tampering with the configuration file as `Decaf` read from the configuration file imme-

diately after if was generated.

`Decaf` was also equipped with a data logger. This data logger records some useful information every time a program is compiled including a snapshot of the program being compiled, the date and time at which the compilation occurred, and the type of error that was detected (if any). A record is created with this information and shipped to a mySQL database. Participants who withdrew from the study were not subject to this data collection. The data collected can be used to gauge student productivity. Student productivity is divided into a number of categories, see Section 4.15 on page 43 for more details.

I originally intended to use a software Frankenstein in the form of the `BlueJ` IDE, modified to use the enhanced compiler `Decaf`, and extended to use the data collection extension for `BlueJ` named `ClockIt`. This combination would have been installed on each of the laboratory computers. I believed `ClockIt` was necessary at the time as I was unfamiliar with `Decaf`'s data collection features. However, I discovered that `ClockIt` would be unsuitable for use in my study after discussing the matter with people close to the project. Without `ClockIt`, there was less of a reason to use `BlueJ` as an editor for programs when `Decaf` was also an editor. As such, I pivoted to a design where `Decaf` was used to meet my enhanced compiler, program editor, and data collection needs.

4.3 Programming Pre-assessment and Anonymous Identifiers

The first day of class for the Winter 2017 semester was January 4th. Participants were supplied with paper slips containing anonymous identifiers on January 9th. Anonymous identifiers were comprised of two components in the following order: a noun and a three digit number. All of the nouns and numbers were unique;

this provided enough redundancy to correct a participant's identifier if minor misspellings were present.

Immediately following the distribution of identifiers was the programming pre-assessment (see Appendix A.5 on page 110). This entrance quiz featured many of the intended learning outcomes for the course but was not graded. The purpose of the pre-assessment was to provide a good indicator of how much prior programming experience each participant had. The questions were approved by both my supervisor and by the instructor for the course. The advantage that the programming pre-assessment quiz has over Questionnaire 1 (see Appendix A.6.1 on page 114) is that it is a more objective measurement of a participants's level of prior programming experience.

Before turning in the pre-assessments, participants were directed to write their anonymous identifiers on the programming pre-assessments and to keep the paper slips for later use. Lastly, participants were instructed to include their name so that their identifier could be recovered if was forgotten it and the paper slip was misplaced.

Some participants were absent and wrote the pre-assessment at a later date. However, late writers had very large advantage over those who wrote the pre-assessment on January 9th. This was due to the lectures in the days following covering many of the concepts that were featured on the pre-assessment. As a result, late writers were not included in statistical analysis as it would have distorted the data.

The programming pre-assessment did not change much between how it was originally imagined and its final draft. Anonymous identifiers were originally proposed to be generated by the participants rather than me. However, I could not guarantee that the participants would include enough redundant, or unique, information to correct minor mistakes in their responses to questionnaires. As such,

this approach was discarded in favour of anonymous identifiers created by me.

4.4 Consent Form and Information Letter

Students were provided with a consent form / information letter (see Appendix A.3 on page 104) and a withdrawal form (see Appendix A.4 on page 108) for the opt-out study on January 25th. Absent students were provided with the same materials at the earliest opportunity in lab and tutorial sessions.

As alluded to in Section 4.1 on page 32, the first version of my study required participants to opt-in. This was latter changed to opt-out due to concerns with small sample sizes. The consent form / information letter changed accordingly.

4.5 Withdrawal Form

At the same time that programming pre-assessments were distributed, all participants were given a dedicated withdrawal form that could be used to end their participation in the study. The advantage of the withdrawal form over other forms of withdrawing is that it could be used at any time. Regardless of the manner and timing in which the withdrawal occurred, participants only needed to include their anonymous identifier and a clear indication of their intention to withdraw by ticking the appropriate box.

The withdrawal form in the first draft of my methodology required participants to include their name when withdrawing from the study. The Research Ethics Board (REB) that evaluated my study was concerned that this presented a perceived or real vulnerability to bias by the researcher. My response to the REB's concern was to remove the requirement for withdrawing participants to include their name. The withdraw from was modified such that only the participants's identifier was required to withdraw.

The REB was also concerned that participants who withdrew could be easily identified when it was time for the class to fill out questionnaires (as they would otherwise be doing nothing during that time). The REB suggested that non-participants should be given something to do on the questionnaire to prevent their identification. I implemented this by including trivia-like questions related to Java on each of the three questionnaires.

4.6 Weekly Assignments

The students' weekly programming assignments began on January 10th and continued for the rest of the semester. `Decaf` was used from the second assignment (January 17th) and onwards. As described in Section 4.2 on page 34, a snapshot was generated and shipped to a database each time one of the participants compiled a program. No snapshots were generated for students who were not participating in the study.

I had little control over the assignments that were completed by CPSC110 students as that was under the domain of the class's instructor. As such, the weekly assignments saw no change between the first and last versions of my thesis.

4.7 Questionnaire 1

I distributed Questionnaire 1 to the students on the week of January 31st (see Appendix A.6.1 on page 114). The purpose of this questionnaire was to allow participants to self-assess both their level of prior programming experience as well as their confidence in their ability to solve programming problems. Students had completed their first programming assignment by this point in time.

Questionnaire 1 was a component of my methodology since the early stages of my thesis. Like the other two questionnaires, Questionnaire 1 saw two notable

changes between what I originally imagined and what was given to participants in my study. One of the major changes is that participants no longer have to include their name when withdrawing from the study via questionnaire. As discussed in Section 4.5 on page 37, the original purpose behind including the names of the participants was to ensure that only participants currently in the study received a bonus credit to final grades. Once this incentive was removed in later revisions of my methodology, there was no further reason to identify withdrawing participants. The second change that occurred to the questionnaires was the addition of activities for participants who had either withdrawn or were withdrawing from the study. This was done at the request of my Research Ethics Board who were concerned that non-participants identified by their peers could suffer from diminished social standing.

4.8 Questionnaire 2

Questionnaire 2 was distributed to the students throughout the week of February 28th (see Appendix A.6.2 on page 116). This questionnaire also asked participants to self-assess their confidence in their ability to solve programming problems. One of the differences between the two questionnaires is that Questionnaire 2 also asks participants to self-assess the level of frustration they experience when fixing programming errors.

Questionnaire 2 changed from beginning to end in precisely the same way as questionnaire 1. For more information on which changes occurred, see Section 4.7 on the preceding page.

4.9 Laboratory Quiz

A laboratory quiz was held during the week of March 28th (see Appendix A.7 on page 120). Much of the data used in statistical analysis was collected during the quiz. The questions were structured to be very similar to the problems students encountered on the lab assignments they had completed previously. Students were instructed to complete as many of the questions as possible in 60 minutes. The quiz had exam-like conditions; students only had access to *`Decaf`* and their programming expertise. The laboratory quiz was primarily for my study, although the students may have found it useful as a review for the final exam.

The original draft for my study included a bonus credit of approximately 3% for participants who were still a part of the study when they completed the laboratory quiz. This incentive was removed in the later versions of my methodology.

4.10 Questionnaire 3

The third and final questionnaire was given to students through the week of April 3rd (see Appendix A.6.3 on page 118). Like the first and second questionnaires, Questionnaire 3 also asked participants to self-assess their confidence in their ability to solve programming problems. Questionnaire 3 also asked students if they would recommend the compiler that they used to other novice programmers. The purpose for this question was to gauge how much appreciation there was towards the compiler.

Like the other two questionnaires, questionnaire 3 saw some changes from what I originally proposed. The details on these changes are described in Section 4.7 on page 38.

4.11 Anonymous Identifier Corrections

After cursory inspection of the data entered into the database, I discovered that there were some discrepancies in the anonymous identifiers used by participants. Namely, that some participants were inconsistent in how they spelled their identifier over the course of the study. In the majority of cases, the participant's mistake was with the numeral portion of their identifier. This issue was corrected in cases where I was very confident I could do so while preserving the integrity of the collected data.

The original design of my research methodology did account for this phenomenon. As described in Section 4.3 on page 35, anonymous identifiers were designed to contain two unique parts. This redundancy would allow me to correct mistakes in the written responses by the participants.

4.12 Participant Grade Scaling

I consulted with the instructor for the course on possible mitigation strategies and he requested that I use the following rules (which were then approved by the chair of the department and by the Research Ethics Board):

- Compute the mean final grade of the control group and the enhanced group.
- If the difference between the means is greater than 3%, apply half the difference to all the students in the group with the lower mean.

I also went one step further than necessary and determined if a significant difference existed at all between the two groups using an independent samples t-test. In the interest of respecting the privacy of the students, I will not be reporting the exact statistics that were computed for grade scaling purposes. However I will say that not only was the mean difference between the groups less than the required

3%, an independent samples t-test showed that there was no significant difference between the control group and the enhanced groups. With these two pieces of evidence, I could say with confidence that no grade scaling was necessary.

The data for this determination was based off of the student's final grades, which were provided by the course instructor. It should be noted that the provided grades were for "students" and not "participants". The students' names were then linked to individual participants with the help of the programming pre-assessment (see Appendix A.5 on page 110) which contained both pieces of information by design. In other words, it was a necessary evil to compromise participant anonymity as otherwise I would not have been able to determine if participants' grades had been unfairly impacted by the study.

In the first version of my research methodology, I did not have a strategy for scaling the participants grades. The REB was concerned that the enhanced compiler may have had a significant effect on the student's grades and they asked that I clearly outline a strategy for mitigating this risk. My response to their concerns, addressed above, was accepted.

4.13 Database Preparation

Once all the data was collected and entered, I began preparing the database so that I could easily (and repeatedly if necessary) pull the data required for the statistical tests I needed to conduct to answer my research questions. This included the process of separating laboratory quiz snapshots from assignment snapshots and organizing snapshots into pairs of consecutive snapshots. In the process of doing so, it became apparent that I had the data required to answer additional research questions that I had not imagined during my proposal.

This component of my thesis was not present in my original methodology. It

was only once I had seen the data in the mySQL database that I was able to determine what needed to be done.

4.14 Laboratory Quiz Snapshot Examination

I examined each pair of snapshots that was captured during the laboratory quiz and categorized each response according to the rubric by Hristova *et al.* [22] (outlined in Section 4.15.2 on the following page). The group that a participant belonged to (*i.e.*, control or enhanced) was intentionally hidden during this process to minimize the potential for bias in applying the rubric.

Additionally, I also examined the last snapshot that was available for each of a participant's programs. The purpose of this examination was to categorize that participant's performance on the laboratory quiz. The final snapshot for each question was placed into one of three categories using the following rubric. The first category, labelled as perfect, was for attempts that not only answered the question but did so without containing any compilation errors. The second category is labelled as imperfect. This category includes attempts at solving a quiz question that contain compilation errors or did not completely answer the question. The final category is for participants who did not even attempt a question.

I came across the rubric by Hristova *et al.*'s rubric early on in my research. It has been a part of my methodology since then. Once I began examining snapshots however, I felt the need to modify the rubric to better suit my needs. The additional categories are described in Section 4.15.2 on the next page.

4.15 Statistical Analysis

This section contains all of the research questions that I answered with the data I had collected over the course of my study. Included is a description of where the

data for the research question originates as well as the statistical test used to analyze it. The alternate and null hypotheses are included for each research question. The statistical test results can be found in Chapter 5 on page 57 and the interpretation of those results in Chapter 6 on page 64.

4.15.1 Choice of α

Academic research often uses an α of 0.05. For my statistical tests, I use an $\alpha = 0.10$ for two reasons. First, thanks to the exploratory nature of study, there is little in the way of historical data to compare my results to. Second, the small sample sizes used in my study have made it difficult to detect small effect sizes with confidence. Using $\alpha = 0.10$ is a reasonable compromise between minimizing the risk of Type I and type II errors while extracting as many useful observations as I can for the data set that I have.

The χ^2-tests in my study often feature multiple tests on the same data in order to compare two proportions out of a set (*i.e.,* X and NOT X, Y and NOT Y) between the control group and the enhanced group. In these cases, a Bonferroni correction has been used in an attempt to be conservative with my findings. As a result, the α used for these tests is not $\alpha = 0.10$ but instead $\alpha = {}^{0.10}/_{N}$, where N is the number of repeated tests done on the same data set.

4.15.2 When Is An Error Considered To Be Successfully Fixed?

Consider a hypothetical assignment where students are learning about the different data types in Java. Part of the assignment involves matching values specified in the assignment with an appropriate data type. If the student tries to assign the value of `1.23` with the statement `int x = 1.23;`, a semantic error will occur as decimals are prohibited for the `int` datatype. If the student then attempts to fix the error by rewriting the statement as `int x = ‘‘1.23’’;`, this would not be con-

sidered a correct fix as the student is no closer to solving the root of the issue. If the student instead rewrites the original statement as `int x = 1;` that would be considered a correct fix as it correctly addresses the problem.

It is possible that students may take a reasonable approach to fix an error, only to make a typing mistake or some other error in the process (which would then cause another syntax or semantic error). In this scenario, the students' attempt would be flagged as partially fixed. By using this categorization, I can determine if the enhanced error messages affect the rate at which new errors were introduced when debugging. I used the rubric created by Hristova *et al.* [22] to assist me in correctly categorizing the fixes attempted by programmers. This rubric features the following categories for responses to the first error message shown to programmers:

- DEL — Wholesale deletion of code, which may include code segments that were not problematic.
- UNR — Unrelated to the cause of the error.
- PART — Partially addresses the cause of the error.
- FIX — Adequately addresses the cause of the error.
- DIFF — Partially or adequately addresses an error that is unrelated to the current error message.

After observing some of the laboratory quiz snapshots, I felt compelled to introduce two additional categories:

- OTHER — Intentionally ignores error message to work on unrelated components of the program.
- NRN — No response needed, for pairs of snapshots where the first snapshot was error-free.

The modified rubric above is what I used to categorize a participants's response to an error message in the later phases of my research.

4.15.2.1 Which Responses To Error Were Productive and Which Were Unproductive?

I deemed each of the categories above to be either productive or unproductive. My reasoning for each category can be found in the list below.

- DEL — Unproductive. The programmer will (eventually) have to retype at least some of what was deleted.
- UNR — Unproductive. The programmer is, at best, no closer to fixing the error that is stopping future progress.
- PART — Productive. Even incomplete progress on an error is more productive than no progress at all.
- FIX — Productive. There is one fewer error stopping the programmer from adding new features to their program (or from being finished entirely).
- DIFF — Productive. Similar to PART and FIX with the only difference being is the the error addressed was not the error that was reported.
- OTHER — Productive. Adding new features to the program is being productive, even if it means procrastinating on fixing an error.
- NRN — Productive. Not needing to fix errors allows the programmer to add new features to their program or to test existing features.

4.15.3 What Are The Phases Of Compilation?

Some compilers such as `javac` operate in phases where a different aspect of the program being compiled is examined in each phase. These compilers are called

multi-pass compilers. An example of some phases that appear in multi-pass compilers would be a lexical phase (creates tokens of the characters that appear in the program), syntactic phase (uses the tokens to create an abstract syntax tree and determines if any syntax rules have been violated), and a semantic phase (determines if any semantic rules have been violated).

Compilers struggle to detect a category of semantic errors called runtime errors. Runtime errors, as the name suggests, occur when attempting to run the program. Integer division by zero is an example of a runtime error (since the result of this division is undefined). Since runtime errors are a type of semantic error, some runtime errors can be detected during the semantic phase. For example, an observant compiler will notice that the statement "`int x = 1 / 0;`" will always result in a runtime error and this information can be conveyed to the programmer. However, "`int y = 1 / z;`" will not always result in a runtime error since the value of `z` (which may be zero) cannot be determined at compile time. Runtime errors (specifically, those that cannot be detected at compile time) are outside the scope of my thesis as I focus on what can be detected a compile time.

While not exactly a compilation phase, programmers should also be concerned about a type of error that I call logical errors. Logical errors occur when a program compiles successfully but does not work as intended. Consider the following example. A programmer wants to calculate the average of two numbers. To do this, an experienced programmer may use the statement "`int x = (a + b) / 2;`". A novice programmer, unaware that Java considers operator precedence, may instead try "`int x = a + b / 2;`". The novice programmer's statement contains a logical error as it does not behave as the programmer intended. Both of the expressions are syntactically and semantically correct, meaning that the compiler will not report any errors. Additionally, neither statement will cause any runtime errors. As a result, it is virtually impossible for compilers to detect logical errors as

they would have to consider the programmer's intentions and this is not possible with the current technology. Like runtime errors, logical errors are also outside the scope of my thesis.

In order to move from the current compilation phase to the next, the compiler must not detect any errors during the current phase. First time programmers are often confused when a multi-pass compiler reports that it has detected more errors after an error was fixed. This phenomenon is a consequence of using a multi-pass compiler on a program that contains errors in more than one compilation phase. For example, if a program contains a lexical error, `javac` will not proceed to the next phase (and thus will not detect any syntactic or semantic errors). When the lexical error is fixed, `javac` will then detect all of the syntactic errors that are present (which may be much higher than the previous number of errors that was reported). This phenomenon also applies to my study in terms of what data is collected; if a participant's program contains errors in multiple phases, only the earliest compilation phase will be recorded. Addressing this limitation would be unfeasible for my study simply due to the nature of how multi-pass compilers work. `Decaf` may mitigate this effect; both the control and enhanced groups were only shown one error at a time. However, the total number of errors detected is also reported which still presents an opportunity for students to become confused.

Up until this point, I have been discussing how typical multi-pass compilers behave. It was necessary for me to determine if `javac` functioned like a typical multi-pass compiler or if it was something entirely different. I read [9] not only for information on how the `javac` compiler behaves but also to determine which compilation phases were necessary to include in my study. I determined that `javac` is not significantly different from the multi-pass compilers discussed above.

The participants of the study also needed to be considered before creating a list of compilation phases to include in my research. As first time Java programmers,

my participants could not be expected to generate errors of all possible types as they simply did not work with enough `Java` features to have the opportunity to do so. For example, most or even all of the programming problems featured on the assignments and the laboratory quiz could be completed with a single public method contained in a single public class. As a result, the students were unlikely to generate errors related to anonymous inner classes. This also means that the participants were unlikely to generates errors for all of the compilation phases. With this in mind, it should not be a surprise that some of the compilation phases discussed in [9] were underpopulated and could not be used with the χ^2-tests that I wanted to use. I addressed this by employing a standard trick of χ^2-tests where underused categories are combined with related categories. The end result of this trick is a smaller number of more populated categories, all of which can be used in χ^2-tests.

The final list of compilation phases that are considered in this thesis, and an example error for each, can be found below.

- Lexical [1] — Failing to close a String literal (*i.e.,* a double quote) that was previously opened.
- Syntactic — Failing to terminate a statement with a semicolon (;).
- Semantic (1) — Attempting to assign a String (*e.g.,* `"hello"`) to an integer variable.
- Semantic (2) — Including a statement that is unreachable and will never execute.
- Okay — No errors detected by `javac`.

[1] This category was underpopulated for the laboratory quiz only. As such, it was combined with the Syntactic category using the trick for χ^2-tests mentioned previously.

The reason that Semantic (1) and Semantic (2) were both kept as separate categories was due to each category containing a sufficient number of errors to be used in statistical testing. It should be noted that the lexical and syntactic categories were combined for the laboratory quiz as they did not have the required counts for the statistical tests I used.

I examined the types of errors that were generated by my participants and choose the category which best represented the error. I am confident I picked the correct categories was there was little ambiguity about which category a type of error belong to.

4.15.4 Why Is Timing Data Only Available For The Laboratory Quiz?

The laboratory quiz featured a heavily controlled environment with minimal distractions. This environment was intended to minimize the time participants spent on doing non-programming related activities (such as conversing with friends, updating their Facebook status, and going to the bathroom) so that they could instead focus on the laboratory quiz. As a result, if 30 seconds had elapsed between two compilations, I could be fairly confident that the participant had spent that entire time working on their program. On the other hand, as I observed first hand, the assignments were not free of distractions and there were few clues to indicate how much time was spent on non-programming activities. In order to preserve the quality of my data, and of my statistical results, I chose to not analyze the timing data of the assignments.

4.15.5 Q1 — Time and Compilations Per Program

For students with similar levels of Prior Programming Experience, do enhanced compilers affect the amount of time and/or number of compilations required to complete a programming problem from start to finish?

As described earlier, a record is created for each compilation. Records were then reorganized to contain pairs of consecutive snapshots (provided that the snapshots were both from the same participant and for the same program). The time taken to complete a task is determined as the sum of the time elapsed between each pair of snapshots. The number of compilations is the number of records that exist for each question.

One flaw with this method is that the time elapsed between opening the IDE and the first compilation is not recorded. Solving this flaw by making a record for opening the IDE introduces a different set of problems. Some students will open the IDE, read through the programming problem, and then start programming. Other students will read through the programming problem first before opening the IDE. Both types of users should finish the assignment in approximately the same amount of time. However, by creating a record when opening the IDE, the time elapsed will be different between these two types of users when no real difference exists.

For both time and compilations, an independent samples t-test will be used to compare the means of the control and enhanced groups for statistical significance.

The null hypothesis, Q1.0, is "The enhanced compiler has no effect on the amount of time/effort required to complete a programming problem." The alternative hypothesis, Q1.A, is "The enhanced compiler has some effect the amount of time/effort required to complete a programming problem."

4.15.6 Q2 — Productivity

For students with similar levels of Prior Programming Experience, do enhanced compilers affect student productivity?

The data for this research question originates from the snapshots captured during the laboratory quiz. Similar to Q1, snapshots were grouped into pairs of consecutive compilations. Each pair was examined by hand to determine if a student's response to an error was productive or unproductive (see Section 4.15.2 on page 44 for more information on how this was done). A χ^2-test test was used to determine if the proportion of productive to unproductive compilations differed between the two groups.

Additionally, the time spent on productive and unproductive activities, both in total as well as on a per compilation basis, is also examined. In both cases, an independent samples t-test is used to determine if significant differences exist between the control group and the enhanced group.

The null hypothesis, Q2.0, is "The enhanced compiler has no effect on student productivity." The alternative hypothesis, Q2.A, is "The enhanced compiler has some effect on student productivity."

4.15.7 Q3 — Phases of Compilation

For students with similar levels of Prior Programming Experience, do enhanced compilers affect the distribution of compilations and/or time across the phases of compilation (including the "Okay" phase)?

The data for Q3 originates from the snapshots collected over the course of the study. Snapshots from the laboratory quiz will be analyzed separately from snapshots collected during assignments. Each snapshot is placed into the compilation phase that best represents the first error that was detected. For example a

`';' expected` error would be considered as a syntactic error. Snapshot that contain no compilation errors are placed into the Okay phase. A χ^2-test test can determine if there is a significant difference between the two groups with regards to how the compilations are distributed.

Furthermore, the time spent on each compilation phase, both in total as well as on a per compilation basis, is also analyzed. An independent samples t-test is used in both of these cases to determine if significant differences between the control group and the enhanced group are present.

The null hypothesis, Q3.0, is "The enhanced compiler has no effect on the distribution of compilations/time among the phases of compilations." The alternative hypothesis, Q3.A, is "The enhanced compiler has some effect on the distribution of compilations/time among the phases of compilations."

4.15.8 Q4 — Frustration When Fixing Errors

For students with similar levels of Prior Programming Experience, do enhanced compilers affect the levels of frustration experienced by students when they are fixing errors in their program?

The data to Q4 will come from participants' answers to Questionnaire 2, Question 1 (see Appendix A.6.2 on page 116). A independent samples t-test will determine if there is a difference in mean frustration experienced by the control group and enhanced group.

The null hypothesis, Q4.0, is "The enhanced compiler has no effect on the frustration experienced by the student when fixing errors." The alternative hypothesis, Q4.A, is "The enhanced compiler has some effect on the frustration experienced by the student when fixing errors."

4.15.9 Q5 — Confidence in Programming Ability

For students with similar levels of Prior Programming Experience, do enhanced compilers affect the level of confidence (and the degree of change for confidence) that students have in their ability to solve programming problems?

The answer to Q5 will come from participants' answers to Question 2 across Questionnaires 1, 2, and 3 (see Appendix A.6.1 on page 114, A.6.2 on page 116, and A.6.3 on page 118). The data will be examined with two different methods. For level of confidence, an independent samples t-test will determine if a significant difference in confidence exists between the two groups at any of the three points in time. For the degree of change for confidence over time, a paired-samples t-test will determine if there is a significant change in confidence between the three points in time for each of the groups.

The null hypothesis, Q5.0, is "The enhanced compiler does not have any effect on the confidence students have in the programming ability." The alternative hypothesis, Q5.A, is "The enhanced compiler does have some effect on the confidence students have in the programming ability."

4.15.10 Q6 — Compiler Appreciation

For students with similar levels of Prior Programming Experience, do enhanced compilers affect the degree to which students recommend the compiler that they used?

The answer to Q6 will come from participants' answers to Questionnaire 3, Question 1 (see Appendix A.6.3 on page 118). An independent samples t-test will determine if there is a difference in the mean "appreciation" participants have for the compiler in the control group and enhanced group.

The null hypothesis, Q6.0, is "The enhanced compiler has no effect on whether

they would recommend the compiler to other students." The alternative hypothesis, Q6.A, is "The enhanced compiler has some effect on whether they would recommend the compiler to other students."

4.15.11 Q7 — Self-assessed Versus Measured PPE

Does a relationship exist between a participant's self-assessed Prior Programming Experience and the results of the programming pre-assessment?

The answer to Q7 will come from participants' answers to Questionnaire 1, Question 1 (see Appendix A.6.1 on page 114) and the programming pre-assessment (see Appendix A.5 on page 110). Pearson's correlation will be used to determine what relationship exists between a participant's self-assessed Prior Programming Experience and their performance on the programming pre-assessment.

The null hypothesis, Q7.0, is "There is no relationship between the two measures of participant Prior Programming Experience." The alternative hypothesis, Q7.A, is "There is some relationship between the two measures of participant Prior Programming Experience."

4.15.12 Q8 — Participant Performance

Does the enhanced compiler affect a participant's performance on programming questions answered under exam-like conditions?

This research question was created at the suggestion of my supervisory committee, who were concerned at the possibility that programmers who learned how to program with the enhanced compiler could become worse programmers than those whose learned with standard compilers such as `javac`. The data for Q8 will come from the participants' snapshots. For performance, I will examine only the last snapshot available for each program that was written for the laboratory quiz. As described in Section 4.14 on page 43, each question will receive a score of per-

fect, imperfect, or not attempted. A χ^2-test will be used to determine if one of the groups has significantly different performance than the other for each of the questions as well as in general.

The null hypothesis, Q8.0, is "The enhanced compiler has no effect on the performance of programming questions answered during a laboratory quiz." The alternate hypothesis, Q8.A, is "The enhanced compiler has some effect on the performance of programming questions answered during a laboratory quiz."

4.15.13 Research Question Which Was Considered But Not Used

I considered investigating the the enhanced compilers effect on the amount of time and compilations spent on each type of error. For both of these measurements, totals across all occurrences of the error were considered as well as average per occurrence. However, I choose not to pursue this research question for two reasons. The first reason is that it would have required a considerable amount of time, too much for one researcher on a tight timeline, to do this for the assignments. The second reason is that, for the laboratory quiz, the sample sizes for each type of error were simply too small to find statistical significant results.

This preliminary research question was broadened to use phases of compilation rather than individual types of errors. Additionally, the component of time was excluded for assignments (though it remained for the laboratory quiz). This modified version of the research question became Q3 (see Section 4.15.7 on page 52).

Chapter 5

Statistical Analysis and Results

This section contains all of the statistical analysis that I have conducted over the course of the thesis. The statistical tests that I have conducted belong to one of two categories. The first category, discussed in Section 5.1, contains that statistical tests that I have done for the questionnaires and the programming pre-assessment that participants completed at various points over the course of the study. The second category, discussed in Section 5.2 on page 59, is the result of the data collected by `Decaf` as the participants completed their programming assignments and laboratory quiz.

5.1 Questionnaires and Programming Pre-assessment

This section contains the statistical tests conducted to analyze the data collected from the questionnaires (see Appendix A.6 on page 114) and programming pre-assessment (see Appendix A.5 on page 110) that were distributed to the participants at various points of time.

5.1.1 Notes On Collected Data

In total, 45 participants wrote the pre-assessment. However, some of the participants were late writers and were not included in statistical analysis for reasons discussed in Section 4.3 on page 35. Since the pre-assessment took place before the second laboratory assignment (and thus before the enhanced group had a chance to use `Decaf`), it was not necessary to split the participants into the control group and the enhanced group.

The questionnaires used a five-point Likert scale, meaning that the maximum possible for these items was 5 and the minimum was 1. The pre-assessment contains fourteen questions. Each answer was graded as 2 if it was correct, 1 if it was partially correct, and 0 if it was incorrect or the answer was missing entirely. In other words, the maximum possible score for the entire pre-assessment was 28 and the minimum was 0. The pre-assessment contained an extreme outlier with a score of 14. This outlier was removed before the data was further analyzed, leaving a sample size of 39.

5.1.2 Statistical Test Results

Descriptive statistics for both the control group and the enhanced group were computed in regards to their performance on the programming pre-assessment, self-assessed Prior Programming Experience, the amount of frustration experienced when fixing errors, the amount of appreciation participants held for the compiler they used, and their confidence at three points in the study. These statistics can be found in Table A.27 on page 100.

The two groups were then compared for each of the items noted previously using an independent-samples `t`-test. The results of the `t`-test can be found in Table A.28 on page 101. The change in confidence over time is examined with the use

of a paired-samples t-test. The results of this t-test can be found in Table A.29 on page 101.

Relationship between self-assessed and measured Prior Programming Experience Lastly the relationship between a participant's self assessed Prior Programming Experience and their performance on the programming pre-assessment was examined by computing Pearson's r. Pearson's correlation determined that there is a moderate relationship between the participants self-assessed Prior Programming Experience and their pre-assessment scores ($r(30) = 0.41$, $p = 0.02$). This test combined the control group and the enhanced group's self-assessed Prior Programming Experience since pre-assessment was already combined. The outlier discussed earlier was also removed for this analysis.

5.2 Decaf Snapshot Analysis

This section focuses on analyzing the snapshots collected by `Decaf`. The snapshots are aggregated into two groups. The first group, labelled as "Quiz" consists of snapshots that occurred during the laboratory quiz. The second group, labelled as "Assignments", consists of all other snapshots that were captured.

The statistical tests fall into one of two categories. The first category, found in Section 5.2.1 on the following page, includes an analysis of the distribution of compile time errors grouped by compilation phase with both group totals and participant averages, the participant's responses to those errors (quiz only), also in terms of group totals and participant averages. The phases of compilation that are considered are detailed in Section 4.15.3 on page 46 The categories of participant responses is outlined in Section 4.15.2 on page 44 with the reasoning for the productive and unproductive categories being discussed in Section 4.15.2.1 on page 46.

The second category, Section 5.2.2 on the following page, includes an analysis of the total amount of time participants spent on each phase of compilation as well as the average time spent per compilation for each phase. Also included is an examination of time that was spent on each type of response to an error. It should be noted that this section is entirely focused on the quiz for reasons discussed in Section 4.15.4 on page 50.

The section concludes with a histogram of the most common compilation errors encountered for both groups for both the assignments and the quiz.

5.2.1 Compilation Error Distribution Analysis

Snapshots were grouped with two different methodologies. The first method was to group by which phase of compilation was an error first detected in. Assignment snapshots were analyzed separately from laboratory quiz snapshots. For assignment analysis, see Section 5.2.1.1. For the analysis of the laboratory quiz, see Section 5.2.1.2 on the next page

5.2.1.1 Assignments

The distribution of the total number of errors encountered per compilation phase in the laboratory assignments can be found in Table A.1 on page 89. Multiple χ^2-tests for independence were used to determine if there was a difference in the distribution of errors between the control group and the enhanced and where exactly the differences, if any, were located. The results from this test can be found in Table A.2 on page 90.

The average number of errors encountered per participant for each compilation phase is in Table A.3 on page 90. An independent samples t-test was conducted to determine if there was a difference between the groups for each phase. The results of that test are held in Table A.4 on page 90.

5.2.1.2 Laboratory Quiz

The distribution of the total number of errors encountered per compilation phase in the quiz can be found in Table A.5 on page 91. Multiple χ^2-tests for independence were used to determine if there was a difference in the distribution of errors between the control group and the enhanced and where exactly the differences, if any, were located. The results from this test can be found in Table A.6 on page 91.

The average number of errors encountered per participant for each compilation phase is in Table A.7 on page 91. An independent samples t-test was conducted to determine if there was a difference between the groups for each phase. The results of that test are held in Table A.8 on page 92.

The distribution of the total number of responses for each response category is captured in Table A.9 on page 92. Multiple χ^2-tests for independence were used to determine if there was a difference in the distribution of errors between the control group and the enhanced and where exactly the differences, if any, were located. The results from this test can be found in Table A.10 on page 93.

The average number of participant responses for each response category is stored in Table A.11 on page 93. An independent samples t-test was conducted to determine if there was a difference between the groups for each response category. The results of that test are held in Table A.12 on page 94.

5.2.2 Timing Data Analysis

Timing data recorded for the participants was examined in multiple ways. The first was examining how participants spent their time with regards to the many phases of compilation. The total amount of time that each group spent on each compilation phase can be found in Table A.13 on page 94. An independent samples t-test was conducted to determine if there was a difference in the above data, with

the results of the test contained in Table A.14 on page 94. The average period of time between compilations for each phase was also analyzed and can be is located in Table A.15 on page 95. Like the previous data, an independent-samples t-test was used to determine if a difference between the two groups existed. The t-test results can be found in Table A.16 on page 95.

The second approach for examining timing data was to look at how much time participants spent the different categories of responses to error messages. The total amount of time each group spent on each response category is contained in Table A.17 on page 95. An independent samples t-test was used to check if the two groups were significantly different for any of the categories. The results of the t-test are in Table A.18 on page 96 The average amount of time that elapsed between compilations for each response category is stored in Table A.19 on page 96 And independent samples t-test was used to determine if there was difference in the average length of time between compilations. The results of this test can be found in Table A.20 on page 97.

5.2.3 Performance Analysis

The performance of each group on each of the laboratory quiz questions is contained in Table A.21 on page 97. Multiple χ^2-tests for independence were conducted to determine if there was a difference between the two groups for any of the questions as well as in general. The number of compilations required to answer each laboratory question was also recorded and this data can be found in Table A.23 on page 98. An independent samples t-test was used to determine if the two groups required a different number of compilations for each of the quiz questions. The results of the t-tests are stored in Table A.24 on page 99. Lastly, the time that participants spent on each questions is stored in Table A.25 on page 99. The test used to determine if the control group and the enhanced group required

different amounts of time for each question was an independent samples t-test. The t-test results can be found in Table A.26 on page 100.

Chapter 6

Discussion

This section contains discussion of the research questions that were outlined in Section 4.15 on page 43 and of the statistical test results that support or refute the theories I formulated before conducting my research. The α used to determine statistical significance is 0.10 unless explicitly stated otherwise, (see Section 4.15.1 on page 44 for an explanation on why α varies). Additionally, the top ten most common errors are discussed in Section 6.9 on page 76.

6.1 Q1 — Time and Compilations Per Program

My interest in the productivity of first time programmers led to the development of research question 1, which examines the novice programmers' ratio of output to input. In the context of Q1, the output is a laboratory quiz question (see Appendix A.7 on page 120) while the input is examined with two approaches. The first approach, detailed in Section 6.1.1 on the next page analyzes the number of compilations required, on average, to complete each question on the laboratory quiz. The second approach, discussed in Section 6.1.2 on the following page examines how the average amount of time that was required to complete a laboratory quiz question. In both cases, the participants' performance is also considered. The

interpretation of these test results is held in Section 6.1.3.

6.1.1 Analysis of Compilations

Table A.24 on page 99 holds the test results that are used in the following analysis. There was insufficient data to use a t-test to analyze Question 1 (imperfect performance), Bonus Question 1 (perfect performance) and Bonus Question 2. For the questions where there was enough data, none of them showed a significant difference between the control group and the enhanced group.

6.1.2 Analysis of Time

The results of the t-test discussed here are contained in Table A.26 on page 100. There was not enough data to conduct a t-test for Question 1 (imperfect performance), Bonus Question 1, and Bonus Question 2 (perfect performance). Question 2, Question 3, and Bonus Question 2 (imperfect performance) do not show a significant difference between the control group and the enhanced group.

The only question which showed a significant difference between the control group and enhanced group is Question 1. This difference is of medium effect size ($0.50 < (d = 0.67) < 0.80$) and is in favour of the control group. The enhanced group did have a noticeable outlier with a time of 680 seconds, which is more than three standard deviations above the mean. However the test was still significant even after the outlier had been removed.

6.1.3 Conclusion

The evidence above suggests that there are minimal differences between the enhanced compiler and the standard compiler on both measures of input that are needed to write programs. In fact, there is some evidence that the enhanced com-

piler actually decreases productivity for simple programs. A possible explanation is that the enhanced group may have lost productivity if its users felt compelled to read the entirety of the longer error messages for easy-to-fix errors.

6.2 Q2 — Productivity

I was curious about what effect an enhanced compiler would have, if any, on the effectiveness of first time programmer responses to error messages and their productivity when fixing errors. Research question 2 explores this topic. This question is approached with two methods. In the first method, I examine the number of compilations for for each response category. The results of this method are in Section 6.2.1. In the second method, I examine the time spent on each response category. This method's results are in Section 6.2.2 on the next page. My conclusions for research question 2 are discussed in Section 6.2.3 on page 68.

6.2.1 Analysis of Compilations

The analysis of compilations examines both proportions and averages. For the comparison of proportions between the control group and the enhanced group, see Section 6.2.1.1. For the comparison of averages, see Section 6.2.1.2 on the following page.

6.2.1.1 Comparison of Proportions

Table A.10 on page 93 has the results of the χ^2-test that are discussed here. A Bonferroni correction is applied to the α for these tests, giving $\alpha = {}^{0.10}/_{7} = 0.014 \approx 0.01$. With this modified α, the χ^2-tests reveal that there a significant difference in for responses in the OTHER (5.00% vs 1.72%) and NRN (16.96% vs 23.44%) categories. When reducing the categories to just two, productive and unproductive, the χ^2-test

shows that enhanced group also has a greater proportion of productive compilations (70.15% vs 77.85%).

6.2.1.2 Comparison of Averages

The table that holds the the independent samples t-test that is relevant to this question is Table A.12 on page 94. Only one response category has a significant result; the OTHER category. Specifically, the enhanced group had fewer compilations where they ignored the most error message to work on other parts of their program. The enhanced compiler had a medium large effect in this instance $(0.50 < (d = 0.70) < 0.80)$.

6.2.2 Analysis of Time

I used two approaches to analyze how much time was spent on each response category between the control group and the enhanced group. The first approach looks at the total time spent on each response category for each of the groups. This approach is discussed in Section 6.2.2.1. The second approach examines the time between individual compilations in each of the response categories. This discussion of this approach can be found in Section 6.2.2.2 on the following page

6.2.2.1 Total Time Per Response Category

The test results that are relevant to this approach are in Table A.18 on page 96. The test reveals that the enhanced group spent significantly less time on the DEL and UNR categories. For the DEL category, the enhanced compiler had a large effect $(0.80 < (d = 0.82))$. The impact on the UNR category was medium in size $(0.50 < (d = 0.60) < 0.80)$. Reducing the categories to just two, productive and unproductive, reveals that the enhanced group also spent significantly less time

overall on unproductive compilations. The effect size here is large ($0.80 < (d = 0.81)$).

6.2.2.2 Time Per Compilation For Each Response Category

Table A.20 on page 97 holds the independent samples t-test that is the focus of this approach. The results show that the enhanced group, on average, spent significantly less time per compilation for the DIFF and FIX categories. The enhanced compiler had a medium impact on the DIFF category ($0.50 < (d = 0.53) < 0.80$) while the FIX category experienced a small change ($0.20 < (d = 0.23) < 0.50$).

6.2.3 Conclusion

It can be stated with confidence that the enhanced compiler improves the proportion of compilations that are productive. This is primarily by accomplished increasing the proportion of compilations where there are no errors to fix. Users of the enhanced compiler made much better use of their time as they spent significantly less time on both types of unproductive responses overall. Students in the enhanced group fixed their errors faster than their peers in the control group.

6.3 Q3 — Phases of Compilation

Research question 3 explores the topic of how the compilations of the participants' programs are distributed through the various phases of compilation (for more information on the phases of compilation, see Section 4.15.3 on page 46). I wanted to determine if an enhanced compiler would have any effect on this distribution as well as how much time is spent on each phase. The laboratory quiz and the assignments were examined separately as the former was a tightly controlled environment and the latter was not. The findings for the assignments can be found

in Section 6.3.1 whereas the findings for the quiz are contained in Section 6.3.2 on the next page. My conclusions on Q3 can be found at Section 6.3.3 on page 72.

6.3.1 Assignments — Analysis of Compilations

The analysis of the number of compilations in each compilation phases for the assignments requires two approaches in the first approach, I compare proportions. This comparison is located in Section 6.3.1.1. The second approach examines the average number of errors encountered by each student for each compilation phase. This approach is held in Section 6.3.1.2 on the following page.

6.3.1.1 Comparison of Proportions

Table A.2 on page 90 has the test results that are of interest. As mentioned in Section 4.15.1 on page 44, a Bonferroni correction is being used. With a total of six separate χ^2-tests being done on the same data set, the α is calculated as $\alpha = {}^{0.10}/_{6} = 0.017 \approx 0.02$.

Even with this conservative α, there is a statistically significant difference between the control group and the enhanced group for every compilation phase (with the sole exception of the lexical phase) as well as in general. This should not be terribly shocking as the χ^2-test is quite sensitive to sample size. With more than 14000 errors considered, it would be more surprising if the tests were not significant. Instead, the focus should be more on the effect size which gives a better idea of just how much difference there is between the groups. The effect size is determined with Cramer's V which considers the degrees of freedom to determine whether an effect is of small, medium, or large size [33]. It turns out that the effect size for comparison of individual compilation phases, including the "Okay" phase where no errors are detected, each has a small effect size (since $V < 0.1$). However, comparing the populations gives a effect size that is in between small and medium

$(0.05 < (V = 0.09) < 0.15$. The enhanced group has a slightly larger proportion of compilations containing syntactic errors (30.80% vs 32.54%) and of compilations containing no errors (32.72% vs 37.87%) while also having slightly smaller proportions for both the first semantic phase (32.40% vs 26.94%) and the second semantic phase (3.59% vs 2.00%).

6.3.1.2 Comparison of Averages

The test results discussed here are derived from Table A.4 on page 90. The results for both the semantic phases are significant with an $\alpha = 0.10$. The enhanced compiler's effect on this issue is of medium size $(0.20 < d < 0.80)$. The Lexical, Syntactic, and Okay phases are not significant.

6.3.2 Laboratory Quiz

The analysis of compilation phases for the laboratory quiz is divided into two parts. In the first part, the number of compilations for each phase is examined. This part can be found in Section 6.3.2.1. In the second part, discussed in Section 6.3.2.2 on the next page the time spent on each compilation phase is discussed.

6.3.2.1 Analysis of Compilations

The analysis of the number of compilations in each compilation phases for the laboratory quiz requires two approaches in the first approach, I compare proportions. This comparison is located in Section 6.3.2.1. The second approach examines the average number of errors encountered by each student for each compilation phase. This approach is held in Section 6.3.2.1 on the next page.

Total Errors Per Compilation Phase Table A.6 on page 91 contains the test results that are discussed here. A Bonferroni correction has been applied to these

test results, giving $\alpha = {}^{0.10}/_{5} = 0.02$. The results show a significant differences between the control group and the enhanced group for just two categories; for compilations containing no errors (21.92% vs 30.02%) as well as general difference between the two populations. Cramer's V, using the guidelines in [33], shows that the enhanced compiler had a small effect on the proportion of error-free to erroneous compilations ($(V = 0.09) < 0.10$) as well as a small-medium effect on the populations in general ($0.06 < (V = 0.10) < 0.17$).

Average Number Of Errors Per Compilation Phase The table holding the tests result discussed in this section is Table A.8 on page 92. None of the tests show a significant difference between the control group and the enhanced group.

6.3.2.2 Analysis of Time

Two methods were used the investigate the time spent on each compilation phase. The first approach looks at the total time dedicated to each compilation phase. The second approach looks at the average time spent on each compilation for each phase. Both approaches are discussed below.

Total Time Per Compilation Phase The tables that contains the independent samples t-test results discussed here is Table A.14 on page 94. The only test that is significant is for the first semantic phase. This test shows that the enhanced group spent significant less time on this type of semantic error. The difference is of medium size ($0.50 < (d = 0.67) < 0.80$).

Average Time Per Compilation For Each Phase Table A.16 on page 95 holds the test results that are relevant here. Like the previous test, the enhanced group requires significantly less time to respond to errors of the first semantic phase. The

effect of the enhanced compiler here is of small-medium size ($0.20 < (d = 0.30) < 0.50$).

6.3.3 Conclusion

The evidence above shows that the enhanced compiler has an effect on how its users' compilations are distributed across the phases of compilation. Perhaps most importantly, the enhanced group has an greater proportion of compilations that contain no compilation errors on both assignments and laboratory quizzes. The enhanced group has a greater proportion of syntactic errors and a reduced proportion of semantic errors on programming assignments. Furthermore, the enhanced compiler reduces the average number of compilations containing errors detected during both semantic phases. Lastly, users of the enhanced group spend significantly less time overall on errors of the first semantic phase which is likely due to significantly quicker responses to these errors.

6.4 Q4 — Frustration When Fixing Errors

The frustration experienced by first time programmers when they fix errors in their programs is the topic of Research question 4. Table A.28 on page 101 contains the results of the independent samples t-test that answers this question. This test shows a significant difference with $\alpha = 0.10$. Using Cohen's d as the effect size reveals the difference is of large size ($(d = 0.75) < 0.80$).

6.4.1 Conclusion

I can say with confidence that the the enhanced compiler greatly reduces the frustration experienced by novice programmers when they attempt to fix errors in their programs.

6.5 Q5 — Confidence in Programming Ability

The confidence of novice programmers, how this confidence changes over time, and what effect the enhanced compiler has on this confidence is the focus of research question 5. I used two different approaches to examine this question. In the first approach, I examined change of confidence over time for the control group and the enhanced groups separately. My findings for this approach can be found in Section 6.5.1. For the second approach, I compared the two groups at each of the points in time that I measured confidence. I discuss this approach more in Section 6.5.2. A summary of my findings for research question 5 are contained in Section 6.5.3 on the next page.

6.5.1 Confidence Change Over Time

The paired samples t-test results that are discussed here originates from Table A.29 on page 101. It is clear that the control group does not experience any significant change over the course of the study. However this observation does apply to the enhanced group which does see change between change between all but the first and second time points.

6.5.2 Comparison of Control And Enhanced Groups

The results of the independent samples t-test that was used to determine if the two groups had significantly different confidence at any point in time is held in Table A.28 on page 101. With none of the tests showing significant results, it cannot be stated with confidence that the two groups had different levels of confidence.

6.5.3 Conclusion

The enhanced compiler had a small effect on how confidence changed over time. Specifically the programmers who used the enhanced compiler `Decaf`, at least in the second half of the semester, experienced a greater increase in confidence in their programming abilities, than their peers who used the commercial compiler `javac`. Despite this finding, there is no evidence that the two groups had different levels of confidence at any of the three points in time where confidence was measured.

6.6 Q6 — Compiler Appreciation

I was curious about whether first time programmers would notice that effects on an enhanced compiler or not. Research question 6 explores this idea. Table A.28 on page 101 is home to the results of the independent samples t-test that answer this question.

With $\alpha = 0.10$, the test shows that there is a significant difference between the two groups. Specifically, members of the enhanced were much more eager ($0.80 < (d = 0.83)$) to recommend the compiler that they used over participants in the control group.

6.6.1 Conclusion

The evidence above shows that first time programmers, even with the little experience they have in the world of programming, are able to appreciate what the enhanced compiler does to make programming more bearable.

6.7 Q7 — Self-assessed Versus Measured PPE

Controlling for the participants' Prior Programming Experience, if any, was a necessary in order to obtain meaningful results for my other research questions. I was also interested in the strength of the relationship between my two measures of Prior Programming Experience and decided to this the focus of research question. The result that is of interest here is originally noted in Section 5.1.2 on page 59.

With $\alpha = 0.10$, Pearson's correlation shows that the relationship between the two measures of Prior Programming Experience is significant. The relationship coefficient $r = 0.41$ means that a moderate positive correlation exists.

6.7.1 Conclusion

The evidence above suggests that study participants can be trusted to accurately report how much Prior Programming Experience they have coming into a class.

6.8 Q8 — Participant Performance

The concern that novices who learn using the enhanced compiler may perform worse programmers than programmers who learn with commercial compilers is explored in research question 8. I used a χ^2-test to determine if this was the case and the results of this test are stored in Table A.22 on page 98. Bonus Question 1 was not attempted by anyone in the control group and could not be examined with a χ^2-test. None of the remaining tests show that there was a significant difference in performance between the two groups on any of the questions as well as in general.

6.8.1 Conclusion

The evidence shows that the enhanced compiler may not have any effect on a participant's performance on laboratory quiz questions. This is corroborated by a previous finding in this thesis; consider the participants' final grades of which laboratory assignments are a sizable portion. As discussed in Section 4.12 on page 41 there was no difference in the participants's final grades between the control group and the enhanced group. Since the enhanced compiler appears to have no effect on participant performance in both examination and assignment settings, it can be said with reasonable confidence that the enhanced compiler has no effect at all on the performance of first time programmers.

6.9 Top Ten Most Common Errors

This section discusses the top ten most common errors that were encountered by the control group and how the enhanced group compares. The top ten most common errors can be found in Appendix A.1 on page 103 for the assignments and Appendix A.2 on page 103 for the laboratory quiz. Section 6.9.1 compares the top ten from the assignments and the laboratory quiz. Section 6.9.2 on the next page examines how the top ten for the assignments compares with similar research by other authors.

6.9.1 Comparison of Assignments and Laboratory Quiz

Six of the ten most common errors on the assignments were also featured in the ten most common laboratory quiz errors. The six error types are "Cannot find symbol", "; missing/expected", "')' expected", "Illegal start of expression", "Not a statement", and "<identifier> expected". The "Cannot find symbol" error was the most common error by far for assignments and laboratory quizzes. I am not

surprised by this finding; the error that I had to help the participants with the most on their assignments was misspellings of variable names which often generates a "Cannot find symbol" error.

6.9.2 Comparison of Assignments and Similar Research

In this section, I compare the top ten most common errors encountered with the most common errors encountered by participants in other studies. It should be noted that the top ten most common errors in my work are unlikely to be identical to the work of other researchers as everyone appeared to use a different set of assignments (with each set of assignments having a different set of errors that could be reasonable generated by participants). The comparisons have been grouped by author. Some of the errors reported in other studies may seem like they are very different from the errors in my top ten but the causes of the error are identical and so I have treated these categories as identical as well. I will report the names of the categories as they appear in the work where they are found. The names of my errors were inspired by the actual errors messages generated by `javac` whereas other researches may have followed a different philosophy for naming their errors.

6.9.2.1 McCall and Kölling

In the study conducted by McCall and Kölling [23], the categories that are used are a little vague but nevertheless there is some overlap between the most common errors that are encountered. The four errors that are included in both of our studies are: "; missing", "variable name written incorrectly" (a match for "cannot find symbol"), "Invalid syntax" (a possible match for "')' expected"), and type mismatch in assignments (a match for "incompatible types").

6.9.2.2 Becker

Becker has conducted multiple studies that feature common errors that students make [4–6]. There is little variation in the top ten most common errors across these studies so I will compare my results to just [5]. The six errors that can be found in both of our top tens are: "Cannot find symbol", "')' expected", "not a statement", "illegal start of expression", "reached end of file while parsing", and "<identifier> expected".

6.9.2.3 Rountree

The categories used in Rountree's study [30] are broader than what I used and only six of their ten most common errors were reported. However there are still four errors that are common between both of our studies. These errors are: "unknown variable" (a match for "cannot find symbol"), "missing ;", "bracket expected" (matches with " ')' expected"), and "illegal start of expression".

6.9.2.4 Jadud

The top ten most common errors recorded in Jadud's study [18] contains seven errors that are also in my top ten. The errors that are in both of our top tens are: "; expected", "unknown variable" (matches with "cannot find symbol"), "bracket expected" (matches with " ')' expected"), "illegal start of expression", "incompatible types", "class or interface expected", and "<identifier> expected".

6.9.2.5 Jackson *et al.*

In Jackson *et al.*'s work, [17], also features seven errors that were in both of the top ten most common errors encountered. The seven errors that match are: "cannot resolve symbol", "; expected", "illegal start of expression", "class or interface expected", "')' expected", "incompatible types", and "not a statement".

6.9.3 Conclusion On Error Types

Overall, there was a large overlap between the errors encountered by the participants in my study and the participants in other studies. Every study, mine included, feature an error in the top ten that is most often caused by misspelling the name of a variable. I would recommend that novice programmers learn to program with a tool that has some sort of built in "spell checker" (at least for the names of variables) as I feel this may be an effective way to address this common error. All but one of the studies that were examined featured missing semicolons as one of the most common errors. It may be worthwhile for future researchers to investigate how to teach novice programmers the importance of semicolons to reduce the frequency with which this punctuation is forgotten.

Chapter 7

Conclusions

This chapter is split into two sections. The first section, Section 7.1, is a brief overview of what occurred over the course of the study. The second section describes my results and recommendations for the future. Additionally, the second section also compares the results of my thesis with similar work. This section is located at Section 7.2 on the next page.

7.1 Summary of Conducted Study

During my time as an undergraduate Computer Science student, as well as during my tenure as a Teaching Assistant, I have had first hand experience with helping novice programmers fix frustrating errors in their programs. My experiences have made me curious on how much productivity is lost among novice programmers who may resort to trial-and-error to fix programming errors. I also wanted to investigate possible approaches that could improve this productivity. To satisfy my curiosity, I conducted a study at the University of Northern British Columbia over the course of the Winter 2018 semester. The class I choose to use for my study was CPSC110 (Introduction to Programming For Non-Majors), which teaches the basics of programming with Java to students who have little to no experience with pro-

gramming. I choose this class over typical CS1 classes because I wanted my study to focus on first time programmers with as little Prior Programming Experience as possible. The class was split into two groups of approximately equal size. The control group used the `javac` compiler while the enhanced group used the `Decaf` enhanced compiler. `Decaf` takes the error messages that are output by `javac` and restructures them so that they are easier to understand for first-time programmers. Every compilation performed by the participants generated a snapshot that was shipped to a database for further analysis. Participants in the study completed a number of study materials including a prior programming pre-assessment, three questionnaires, and a laboratory quiz. My findings are presented below, along with recommendations that follow these conclusions and potential future work in related areas.

7.2 Conclusions and Recommendations

The enhanced compiler was successful at reducing the proportion of compilations that contained compilation errors generated in both assignment and laboratory quiz settings. In other words, a greater proportion of compilations were dedicated to adding new features to programs instead of fixing errors. I feel that this impact alone is enough to recommend to universities like UNBC that they use enhanced compilers in the teaching of Computer Science and the basics of programming to students. There is potential for future work in this area; namely, more studies need to be conducted to determine if this finding holds true for students with more programming experience (such as second and third year students). My findings corroborate Becker's as he also found that users of enhanced compilers generate less errors than users of commercial compilers [4, 5].

The enhanced compiler has an impact on the distribution of compilations across

the various phases of compilation for both assignments and laboratory quizzes. Specifically, the novice programmers who used the enhanced compiler were slightly more likely to generate syntactic errors and slightly less likely to generate semantic errors on their assignments. Even with the much smaller number of errors that were considered for the laboratory quiz, it can be stated with confidence that the enhanced compiler was effective at addressing errors of the first semantic phase (which includes errors related to type checking). The additional information in the enhanced compiler's error messages may have helped its users understand the causes of the semantic errors which in turn would have helped these programmers avoid these errors in future compilations. `Decaf`'s users were able to address these errors faster when they did occur than users of `javac`, which resulted in the enhanced group spending significantly less time overall on these errors. As for the syntactic errors, the additional information may have been less helpful for some of the errors in this category. For example, programmers in both groups were serial forgoers of semicolons despite being well aware that statements in Java must end with this punctuation. Future work can investigate this change in distribution and whether it has beneficial, harmful, or neutral impact on novice programmers. In my thesis, syntax errors outnumbered semantic errors (with the exception of the control group on assignments, where semantic errors were slightly more common). This agrees with Lewis and Mulley's results [21]; namely, that novices generate a greater proportion of syntax errors while experienced programmers have a greater proportion of semantic errors.

The enhanced compiler was successful at reducing the frustration that novice programmers experience when they fix errors in their programs. I have previously discussed in Section 3.2 on page 28 why I believe it is beneficial to reduce the frustration encountered by novice programmers. This finding is another reason why I recommend the use of enhanced compiles in CS1 classrooms for universities

like UNBC. Becker's findings agree with my own; we both found that enhanced compilers are successful at making compiler errors less frustrating for students [4].

The enhanced compiler has mixed results on the productivity of novice programmers on laboratory quizzes. For very simple programs, there is evidence that the enhanced compiler can actually reduce the productivity of its users. This may be a consequence of the enhanced compilers longer error messages. Future work on enhanced compilers can focus on the effect of message length on the productivity of CS1 students. The enhanced compiler did not have a discernible effect on the time nor compilations required to complete the more sophisticated quiz questions. Users of the enhanced group had a greater proportion of compilations that were productive and spent significantly less time overall on both types of unproductive activities. There is also evidence that students who used the enhanced compiler were able to make progress, including complete fixes, on errors faster than students in the control group. This is finding is yet another reason why I recommend the use of enhanced compilers in the teaching of CS1. It should be noted that my study only measured productivity on laboratory quiz questions. Future studies can focus on the effect of the enhanced compiler, if any, on novice programmers' productivity on assignments. This would require a scheme to minimize the effect of the participants being distracted by causes not related to programming. Other work in the future can examine different approaches for improving the productivity of novice programmers. My results run contrary to Nienaltowski *et al.*'s [26] who found that "long form" error messages, like the ones used by `Decaf`, did not impact response times relative to the "short form" error messages produced by `javac`.

The enhanced error messages generated by `Decaf` were less likely to be ignored by its users than novice programmers who had used `javac`. This is beneficial as procrastinating on an error fix can cause trouble later on in the development of

a program. Specifically, the compiler cannot consistently determine if any code segments after the first error, including new features, are erroneous or not. As such, novice programmers may waste time and effort if they attempt to fix errors in the new code segments if those errors don't actually exist and I believe this habit should be discouraged early on in a novice programmer's studies. The evidence suggests that enhanced compilers can be used to achieve this. I was unable to find any historical data on how enhanced compilers affect the proportion of responses which ignore the reported error message. Future studies should examine the impact of the enhanced compiler on the rate of error messages that are ignored. In doing so, it will become more clear if this observation exists in more classrooms than just those in western Canadian universities.

The enhanced compiler that was used did not have any affect on the academic performance of novice programmers for both assignments and laboratory quizzes. This finding may be useful for universities that are considering using an enhanced compiler but are concerned their effects in academic achievement. I was unable to find any historical data on the effect of enhanced compilers on academic performance nor the effect on attrition rates. Rountree attempted to address attrition rates with *ClockIt* but provides no data nor statistics on the effectiveness of this tool [30]. I believe that future work in this area is important and the field of Computer Science would benefit from examining the effects of enhanced compilers on not only academic achievement but also student attrition rates over a number of years.

Students who used the *Decaf* enhanced compiler were much more eager to recommend it to other first time programmers than students who used the *javac* commercial compiler. It appears that even first time programmers are capable of noticing and appreciating more helpful error messages. Becker also examined if there was a significant difference in eagerness to recommend the enhanced com-

piler [4]. It is worthwhile to compare my results with Becker's as the format of our studies were similar with regards to how error messages were presented to the participants. (the control group was presented with `javac` messages while the the enhanced/intervention group was presented with both `javac` and `Decaf` messages). Becker used two approaches for this examination. In the first approach, Becker used a independent samples t-test to compare the groups' answers to a questionnaire asking "Would you recommend `Decaf` to someone who wants to learn Java but has never programmed before?". The results of the test suggest that there is no conclusive evidence that the groups were significantly different in this regard. It should be noted that my test results on this question were almost identical to Becker's (0.06 vs 0.07) and that Becker's results are significant with the α that I used ($\alpha = 0.10$). The participants in Becker's study also had the option of adding comments to their questionnaire answers. Becker's second approach focuses on these comments. It should be noted that Becker's intervention group left numerous comments recommending `Decaf` while the control group did not leave any. This can be taken as evidence that the intervention group was more eager to recommend `Decaf` than the control group. In light of the above comparisons, it can be stated that my findings agree with Becker's.

Many of the tests that were performed during the study, especially for the laboratory quiz, featured only a small number of participants and were lacking in statistical power as a result. It would be beneficial to repeat these tests with a future study that has either more participants or more laboratory quizzes to examine. The repeat tests would have increased statistical power and would provide more conclusive results on how the enhanced compiler affects laboratory quizzes.

Bibliography

[1] M.D. Balso and A.D. Lewis, First steps: A guide to social research, Nelson Thomson Learning, 2005.

[2] Titus Barik, Kevin Lubick, Samuel Christie, and Emerson Murphy-Hill, How developers visualize compiler messages: A foundational approach to notification construction, Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on, IEEE, 2014, pp. 87–96.

[3] Titus Barik, Jim Witschey, Brittany Johnson, and Emerson Murphy-Hill, Compiler error notifications revisited: An interaction-first approach for helping developers more effectively comprehend and resolve error notifications, Companion Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 536–539.

[4] B.A. Becker, An exploration of the effects of enhanced compiler error messages for computer programming novices, Master's thesis, Dublin Institute of Technology, November 2015.

[5] ______, An effective approach to enhancing compiler error messages, Proceedings of the 47th ACM Technical Symposium on Computing Science Education, ACM, March 2016, pp. 126–131.

[6] B.A. Becker and C Mooney, Categorizing compiler error messages with principal component analysis, 12th China-Europe International Symposium on Software Engineering Education (2016).

[7] Philip J Brown, Error messages: The neglected area of the man/machine interface, Communications of the ACM **26** (1983), no. 4, 246–249.

[8] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral, Syntax errors just aren't natural: Improving error reporting with language models, Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, 2014, pp. 252–261.

[9] David Erni and Adrian Kuhn, The hackers guide to javac, University of Bern, Bachelor's thesis, supplementary documentation (2008).

[10] Franz Faul, Edgar Erdfelder, Axel Buchner, and Albert-Georg Lang, Statistical power analyses using g* power 3.1: Tests for correlation and regression analyses, Behavior research methods **41** (2009), no. 4, 1149–1160.

[11] Franz Faul, Edgar Erdfelder, Albert-Georg Lang, and Axel Buchner, G* power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences, Behavior research methods **39** (2007), no. 2, 175–191.

[12] James B. Fenwick, Jr., Cindy Norris, Frank E. Barry, Josh Rountree, Cole J. Spicer, and Scott D. Cheek, Another look at the behaviors of novice programmers, SIGCSE Bull. **41** (2009), no. 1, 296–300.

[13] Thomas Flowers, Curtis A Carver, and James Jackson, Empowering students and building confidence in novice programmers through gauntlet, Frontiers in Education, 2004. FIE 2004. 34th Annual, IEEE, 2004, pp. T3H–10.

[14] D. Freedman, R. Pisani, and R. Purves, Statistics, International student edition, W.W. Norton & Company, 2007.

[15] Mark Guzdial, What's the best way to teach computer science to beginners?, Commun. ACM **58** (2015), no. 2, 12–13.

[16] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri, Identifying and correcting java programming errors for introductory computer science students, ACM SIGCSE Bulletin, vol. 35, ACM, 2003, pp. 153–156.

[17] James Jackson, MJ Cobb, and Curtis Carver, Identifying top java errors for novice programmers, Frontiers in Education Conference, vol. 35, STIPES, 2005, p. T4C.

[18] Matthew C Jadud, A first look at novice compilation behaviour using bluej, Computer Science Education **15** (2005), no. 1, 25–40.

[19] Matthew C. Jadud, Methods and tools for exploring novice compilation behaviour, Proceedings of the Second International Workshop on Computing Education Research (New York, NY, USA), ICER '06, ACM, 2006, pp. 73–84.

[20] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg, The bluej system and its pedagogy, Computer Science Education **13** (2003), no. 4, 249–268.

[21] Stuart Lewis and Gaius Mulley, A comparison between novice and experienced compiler users in a learning environment, ACM SIGCSE Bulletin, vol. 30, ACM, 1998, pp. 157–161.

[22] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi, Measuring the effectiveness of error messages designed for novice programmers, Proceedings of the 42nd ACM technical symposium on Computer science education, ACM, 2011, pp. 499–504.

[23] Davin McCall and Michael Kölling, Meaningful categorisation of novice programmer errors, 2014 IEEE Frontiers in Education Conference (FIE) Proceedings, IEEE, 2014, pp. 1–8.

[24] Redford Williams M.D, How does stress differ from frustration? - abc news, 2008.

[25] Jonathan P Munson and Elizabeth A Schilling, Analyzing novice programmers' response to compiler error messages, Journal of Computing Sciences in Colleges **31** (2016), no. 3, 53–61.

[26] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer, Compiler error messages: What can help novices?, SIGCSE Bull. **40** (2008), no. 1, 168–172.

[27] Cindy Norris, Frank Barry, James B. Fenwick Jr., Kathryn Reid, and Josh Rountree, Clockit: Collecting quantitative data on how beginning software developers really work, SIGCSE Bull. **40** (2008), no. 3, 37–41.

[28] University of Northern British Columbia, Facts and statistics — university of northern british columbia, 2017.

[29] Daphne E Pedersen, Jenelle Swenberger, and Katlyn E Moes, School spillover and college student health, Sociological Inquiry **87** (2017), no. 3, 524–546.

[30] Joshua Joel Rountree, Clockit: Monitoring and visualizing student software development profiles, Ph.D. thesis, Appalachian State University, 2010.

[31] V.S. Thatcher, A. McQueen, and N. Webster, The new webster encyclopedic dictionary of the english language, Avenel Books, 1984.

[32] Paul T. Wong, Frustration, exploration, and learning., Canadian Psychological Review/Psychologie canadienne **20** (1979), no. 3, 133 – 144.

[33] Charles Zaiontz, Effect size for chi-square test — real statistics using excel, 2013.

[34] Anna Zajacova, Scott M Lynch, and Thomas J Espenshade, Self-efficacy, stress, and academic success in college, Research in higher education **46** (2005), no. 6, 677–706.

Appendix A

Tables and Forms

A.1 Tables

Group	*Control*	*Enhanced*
Phase	Count	Count
Lexical	40	44
Syntactic	2442	2167
Semantic (1)	2569	1794
Semantic (2)	285	133
Okay	2592	2522

Table A.1: Assignments — Total Number Of Errors Encountered Per Compilation Phase. Okay = No Errors Detected

Group	*Control*	*Enhanced*				
Phase	Count	Count	χ^2	df	sig	Cramer's V
Lexical	40	44	1.54	1	0.21	0.01
Not Lexical	7888	6616				
Syntactic	2442	2167	5.04	1	0.02	0.02
Not Syntactic	5486	4493				
Semantic (1)	2569	1794	49.4	1	<0.01	0.06
Not Semantic (1)	5359	4866				
Semantic (2)	285	133	33.20	1	<0.01	0.05
Not Semantic (2)	7643	6527				
Okay	2595	2522	42.55	1	<0.01	0.05
Not Okay	5336	4138				
Population			101.04	4	<0.01	0.08

Table A.2: Assignments — Total Number Of Errors Encountered Per Compilation Phase — χ^2-test Results. Okay = No Errors Detected

Group	*Control*		*Enhanced*	
Phase	Mean	SD	Mean	SD
Lexical	2.00	2.53	2.10	3.19
Syntactic	122.10	78.64	103.19	67.83
Semantic (1)	128.45	92.23	85.43	49.51
Semantic (2)	14.25	13.51	6.33	4.53
Okay	129.60	112.04	120.10	67.35

Table A.3: Assignments — Average Number Of Errors Encountered Per Participant For Each Compilation Phase. Okay = No Errors Detected

Phase	t	df	sig	Cohen's d
Lexical	–0.11	39	0.92	0.03
Syntactic	0.83	39	0.41	0.26
Semantic(1)	0.14	39	0.07	0.58
Semantic(2)	2.49	23.04	0.02	0.79
Okay	0.33	39	0.74	0.10

Table A.4: Assignments — Average Number Of Errors Encountered Per Participant For Each Compilation Phase — t-test Results. Okay = No Errors Detected

Group	*Control*	*Enhanced*
Phase	Count	Count
Lexical	3	0
Syntactic	227	232
Semantic (1)	160	131
Semantic (2)	9	10
Okay	112	160
Participants	17	18

Table A.5: Laboratory Quiz — Total Number Of Errors Encountered Per Compilation Phase. Okay = No Errors Detected

Group	*Control*	*Enhanced*				
Phase	Count	Count	χ^2	df	sig	Cramer's V
Lexical + Syntactic	230	232	0.23	1	0.63	0.01
Not Lexical + Syntactic	281	301				
Semantic (1)	160	131	5.88	1	0.02	0.08
Not Semantic (1)	351	402				
Semantic (2)	9	10	0.02	1	0.89	< 0.01
Not Semantic (2)	502	523				
Okay	112	160	8.89	1	< 0.01	0.09
Not Okay	399	373				
Population			10.96	3	0.01	0.10

Table A.6: Laboratory Quiz — Total Number Of Errors Encountered Per Compilation Phase — χ^2-test Results. Okay = No Errors Detected

Group	*Control*		*Enhanced*	
Phase	Mean	SD	Mean	SD
Lexical + Syntactic	13.53	7.26	12.89	8.22
Semantic(1)	9.41	6.70	7.28	5.08
Semantic(2)	0.53	1.28	0.56	1.25
Okay	6.59	5.08	8.89	6.61

Table A.7: Laboratory Quiz — Average Number Of Errors Encountered For Each Compilation Phase. Okay = No Errors Detected

Phase	t	df	sig	Cohen's d
Lexical + Syntactic	0.24	33	0.81	0.08
Semantic (1)	1.07	33	0.29	0.36
Semantic (2)	−0.61	33	0.95	0.02
Okay	−1.15	33	0.26	0.39

Table A.8: Laboratory Quiz — Average Number Of Errors Encountered For Each Compilation Phase — t-test Results. Okay = No Errors Detected

Group	*Control*	*Enhanced*
Response	Count	Count
DEL	20	11
UNR	120	92
PART	55	66
DIFF	21	17
FIX	143	162
OTHER	23	8
NRN	78	109
Productive	320	362
Unproductive	140	103
Participants	17	18

Table A.9: Laboratory Quiz — Distribution of Participant Responses to Error Messages

Group	*Control*	*Enhanced*				
Phase	Count	Count	χ^2	df	sig	Cramer's V
DEL	20	11	2.81	1	0.09	0.06
Not DEL	440	454				
UNR	120	92	5.20	1	0.02	0.07
Not UNR	340	373				
PART	55	66	1.02	1	0.31	0.03
Not PART	405	399				
DIFF	21	17	0.49	1	0.49	0.02
Not DIFF	439	448				
FIX	143	162	1.47	1	0.22	0.04
Not FIX	317	303				
OTHER	23	8	7.68	1	<0.01	0.09
Not OTHER	437	457				
NRN	78	109	6.03	1	0.01	0.08
Not NRN	382	356				
Productive	320	362	8.19	1	<0.01	0.09
Unproductive	140	103				

Table A.10: Laboratory Quiz — Distribution of Participant Responses to Error Messages — χ^2-test Results

Group	*Control*		*Enhanced*	
Response	Mean	SD	Mean	SD
NRN	4.59	4.27	6.06	5.50
DEL	1.18	1.24	0.61	0.70
UNR	7.06	3.99	5.11	5.71
DIFF	1.24	1.64	0.94	1.55
PART	3.24	2.84	3.67	5.65
FIX	8.41	4.57	9.00	5.03
OTHER	1.35	1.69	0.44	0.71
Productive	18.82	9.17	20.11	11.73
Unproductive	8.24	4.78	5.72	5.91

Table A.11: Laboratory Quiz — Average Number of Compilations Per Participant For Each Response Category

Response	t	df	sig	Cohen's d
NRN	–0.88	33	0.39	0.30
DEL	1.68	24.95	0.11	0.57
UNR	1.16	33	0.25	0.30
DIFF	1.80	33	0.25	0.13
PART	0.54	33	0.59	0.10
FIX	–0.36	33	0.78	0.12
OTHER	2.05	21.13	0.05	0.70
Productive	–0.36	33	0.72	0.12
Unproductive	1.40	33	0.17	0.47

Table A.12: Laboratory Quiz — Average Number of Compilations Per Participant For Each Response Category — t-test Results

Group	*Control*		*Enhanced*	
Phase	Mean	SD	Mean	SD
Lexical + Syntactic	751.06	726.50	588.61	670.26
Semantic(1)	610.12	542.08	327.17	245.76
Semantic(2)	21.71	23.67	23.67	62.66
Okay	649.41	571.81	818.22	709.28

Table A.13: Laboratory Quiz — Total Time Spent (in seconds) On Each Compilation Phase. Okay = No Errors Detected

Phase	t	df	sig	Cohen's d
Lexical + Syntactic	0.69	33	0.50	0.23
Semantic (1)	1.97	22.03	0.06	0.67
Semantic (2)	–0.09	33	0.93	0.04
Okay	–0.77	33	0.44	0.26

Table A.14: Laboratory Quiz — Total Time Spent (in seconds) On Each Compilation Phase — t-test Results. Okay = No Errors Detected

Group	*Control*		*Enhanced*	
Phase	Mean	SD	Mean	SD
Lexical + Syntactic	59.11	116.75	48.82	120.90
Semantic (1)	65.23	54.76	45.30	57.42
Semantic (2)	52.71	51.15	47.33	25.93
Okay	141.54	140.43	135.12	190.56

Table A.15: Laboratory Quiz — Average Time Spent (in seconds) Per Compilation For Each Compilation Phase. Okay = No Errors Detected

Phase	t	df	sig	Cohen's d
Lexical + Syntactic	0.90	431	0.37	0.09
Semantic (1)	2.27	287	0.02	0.30
Semantic (2)	0.28	14	0.79	0.13
Okay	0.25	185	0.80	0.04

Table A.16: Laboratory Quiz — Average Time Spent (in seconds) Per Compilation For Each Compilation Phase — t-test Results. Okay = No Errors Detected

Group	*Control*		*Enhanced*	
Response	Mean	SD	Mean	SD
NRN	649.41	571.81	818.22	709.28
DEL	83.13	101.06	21.78	31.87
UNR	307.76	222.15	174.72	221.28
DIFF	51.82	91.21	25.06	45.05
PART	234.76	379.03	284.22	389.79
FIX	455.06	299.08	317.67	198.16
OTHER	250.35	239.54	116.00	253.44
Productive	1641.41	853.51	1561.17	811.81
Unproductive	390.88	254.35	196.50	222.73

Table A.17: Laboratory Quiz — Total Time Spent (in seconds) On Each Response Category

Response	t	df	sig	Cohen's d
NRN	−0.77	33	0.45	0.26
DEL	2.39	18.99	0.03	0.82
UNR	1.77	33	0.09	0.60
DIFF	1.11	33	0.28	0.37
PART	−0.38	33	0.71	0.13
FIX	1.61	33	0.12	0.54
OTHER	1.20	27.2	0.23	0.54
Productive	0.29	33	0.78	0.10
Unproductive	2.40	33	0.02	0.81

Table A.18: Laboratory Quiz — Total Time Spent (in seconds) On Each Response Category — t-test Results

Group	*Control*		*Enhanced*	
Response	Mean	SD	Mean	SD
NRN	141.54	140.43	135.12	190.56
DEL	70.65	88.43	35.64	31.27
UNR	43.60	41.04	34.18	41.01
DIFF	41.95	34.17	26.53	17.44
PART	72.56	145.39	77.52	185.69
FIX	54.10	101.68	35.30	51.61
OTHER	185.04	180.47	261.00	199.08
Productive	87.20	131.66	77.63	147.35
Unproductive	47.46	51.00	34.34	39.95

Table A.19: Laboratory Quiz — Average Time Spent (in seconds) Per Compilation For Each Response Category

Response	t	df	sig	Cohen's d
NRN	0.25	185	0.80	0.04
DEL	1.60	26.06	0.12	0.53
UNR	1.67	210	0.10	0.23
DIFF	1.80	30.94	0.08	0.57
PART	–0.16	119	0.87	0.03
FIX	2.00	204.61	0.05	0.23
OTHER	–1.00	29	0.33	0.40
Productive	0.89	680	0.37	0.07
Unproductive	2.25	240.02	0.03	0.29

Table A.20: Laboratory Quiz — Average Time Spent (in seconds) Per Compilation For Each Response Category — t-test Results

Group	*Control*			*Enhanced*		
Question	P	I	N	P	I	N
Question 1	15	1	1	17	1	0
Question 2	7	10	0	6	12	0
Question 3	3	6	8	3	7	8
Bonus 1	0	0	17	2	1	15
Bonus 2	0	1	16	5	2	11
Total	25	18	42	33	23	34

Table A.21: Laboratory Quiz — Participant Performance On Laboratory Quiz. P = Perfect, I = Imperfect, N = Not Attempted

Group		*Control*	*Enhanced*				
Question	Performance	Count	Count	χ^2	df	sig	Cramer's V
Question 1	P	15	17	0.01	1	0.93	0.01
	I	1	1				
Question 2	P	7	6	0.23	1	0.63	0.08
	I	10	12				
Question 3	P	3	3	0.02	1	0.88	0.03
	I	6	7				
Bonus 1	P	0	2	—	—	—	—
	I	0	1				
Bonus 2	P	0	5	1.90	1	0.17	0.49
	I	1	2				
All Questions	P	25	33	0.01	1	0.93	0.01
	I	18	23				

Table A.22: Laboratory Quiz — Participant Performance On Laboratory Quiz — χ^2-test Results. P = Perfect, I = Imperfect

Group		*Control*		*Enhanced*	
Question	Performance	Mean	SD	Mean	SD
Question 1	P	2.87	2.00	4.35	3.53
	I	7.00	—	6.00	—
Question 2	P	14.29	13.77	10.50	4.42
	I	21.40	10.88	16.17	9.68
Question 3	P	22.00	4.00	11.67	8.33
	I	10.83	9.39	10.29	9.27
Bonus 1	P	—	—	19.00	16.97
	I	—	—	7.00	—
Bonus 2	P	—	—	5.40	2.88
	I	13.00	—	8.00	4.24

Table A.23: Laboratory Quiz — Number Of Compilations For Each Laboratory Quiz Question. P = Perfect, I = Imperfect

Question	Performance	t	df	sig	Cohen's d
Question 1	P	−1.44	30	0.16	0.52
	I	—	—	—	—
Question 2	P	0.69	7.40	0.51	0.37
	I	1.19	20	0.25	0.51
Question 3	P	1.94	4	0.13	1.58
	I	0.11	11	0.92	0.06
Bonus 1	P	—	—	—	—
	I	0.96	1	0.51	1.67
Bonus 2	P	—	—	—	—
	I	—	—	—	—

Table A.24: Laboratory Quiz — Number Of Compilations For Each Laboratory Quiz Question — t-test Results. P = Perfect, I = Imperfect

Group		*Control*		*Enhanced*	
Question	Performance	Mean	SD	Mean	SD
Question 1	P	41.40	58.04	126.94	170.81
	I	121.00	—	436.00	—
Question 2	P	920.00	695.23	694.17	351.07
	I	1604.90	891.27	1161.58	848.28
Question 3	P	1765.00	352.26	762.00	739.28
	I	842.17	560.10	633.57	591.12
Bonus 1	P	—	—	1107.00	256.39
	I	—	—	235.00	—
Bonus 2	P	—	—	202.60	173.23
	I	944.00	—	378.50	406.59

Table A.25: Laboratory Quiz — Time Spent (in seconds) On Each Laboratory Quiz Question. P = Perfect, I = Imperfect

Question	Performance	t	df	sig	Cohen's d
Question 1	P	−1.94	20.07	0.07	0.67
	I	—	—	—	—
Question 2	P	0.75	9.13	0.47	0.28
	I	1.19	20	0.25	0.51
Question 3	P	2.12	4	0.10	1.73
	I	0.65	11	0.53	0.36
Bonus 1	P	—	—	—	—
	I	—	—	—	—
Bonus 2	P	—	—	—	—
	I	1.14	1	0.46	1.97

Table A.26: Laboratory Quiz — Total Time Spent (in seconds) On Each Laboratory Quiz Question — t-test Results. P = Perfect, I = Imperfect

Group	*Control*		*Enhanced*	
Item	Mean	SD	Mean	SD
Self-Assessed PPE	1.50	0.79	1.63	0.83
Measured PPE	5.06	3.46	4.44	1.98
Frustration	3.67	0.99	2.88	1.11
Compiler Appreciation	3.44	0.88	4.07	0.62
Confidence (T1)	2.72	0.83	2.53	0.70
Confidence (T2)	2.58	0.79	2.76	0.66
Confidence (T3)	3.00	0.71	3.43	0.65

Table A.27: Questionnaires and Programming Pre-assessment — Descriptive Statistics. PPE = Prior Programming Experience

Item	t	df	sig	Cohen's d
Self-assessed PPE	−0.49	35	0.62	0.16
Measured PPE	0.65	27.05	0.52	0.22
Frustration	2.00	25.5	0.06	0.75
Compiler Appreciation	−2.01	21	0.06	0.83
Confidence (T1)	0.78	35	0.44	0.25
Confidence (T2)	−0.67	27	0.51	0.25
Confidence (T3)	−1.50	21	0.15	0.63

Table A.28: Questionnaires and Programming Pre-assessment — t-test Results. PPE = Prior Programming Experience

Group	Time Interval	Mean Difference	SD	t	df	sig
Control	T1 → T2	0.00	1.18	0.00	10	1.00
	T2 → T3	−0.17	0.41	−1.00	5	0.36
	T1 → T3	−0.13	0.99	−0.36	7	0.73
Enhanced	T1 → T2	−0.13	0.72	−0.70	15	0.50
	T2 → T3	−0.54	0.66	−2.94	12	0.01
	T1 → T3	−0.77	0.93	−2.99	12	0.01

Table A.29: Change in Confidence Over Time Within Each Group — Paired Sample t-test Results

A.2 Diagrams

Figure A.1: Assignments — Top Ten Most Common Errors

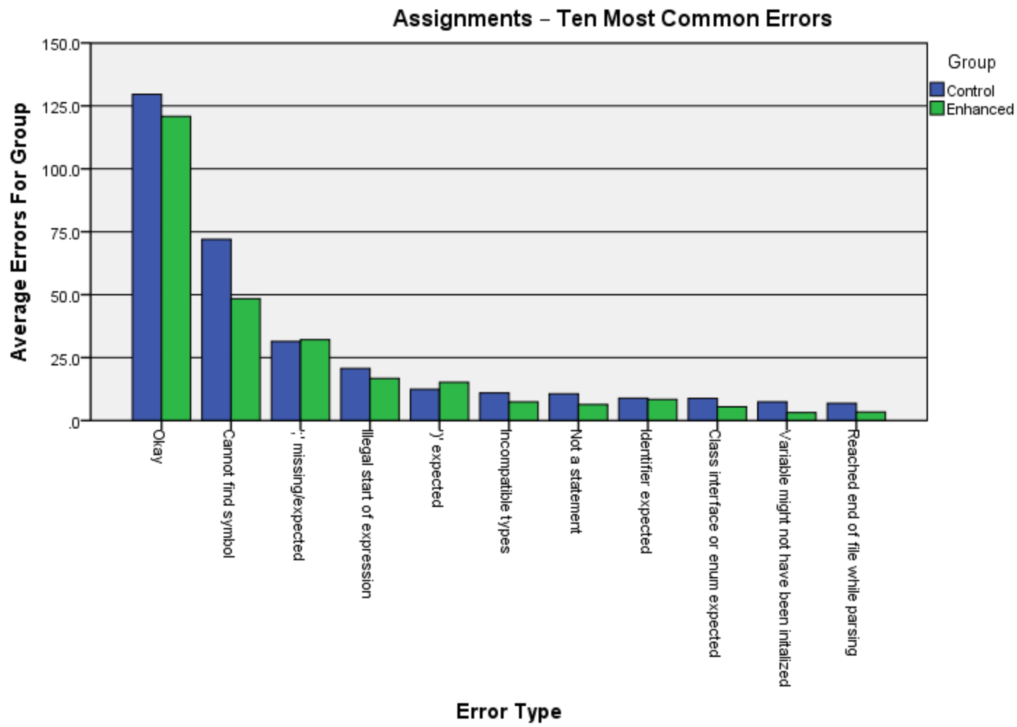
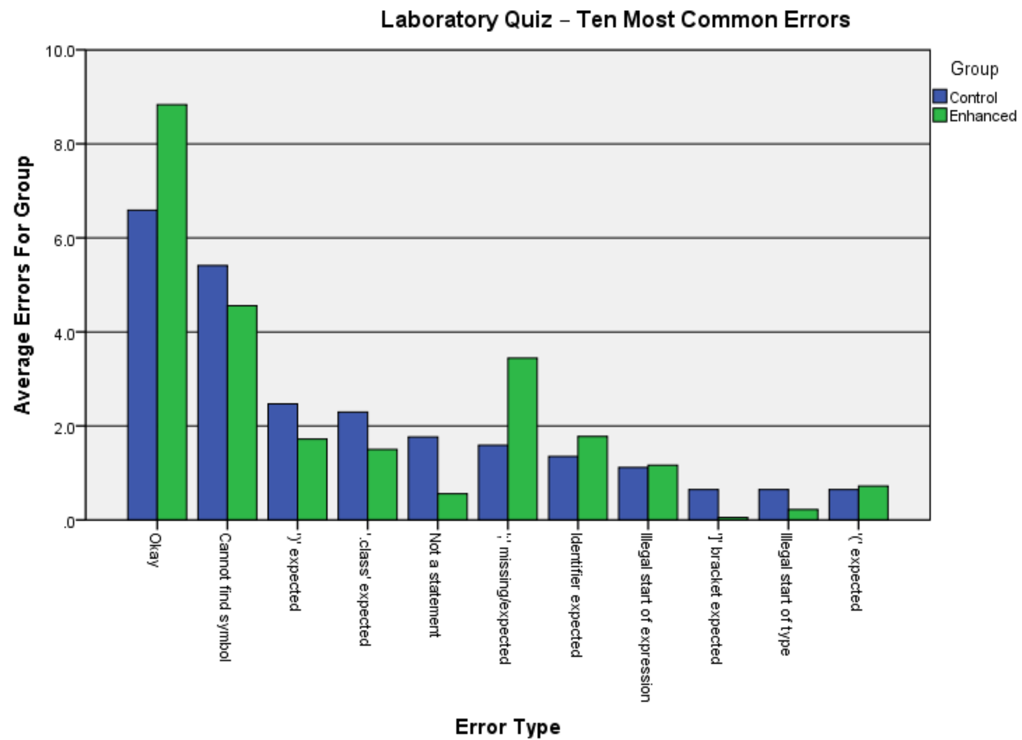


Figure A.2: Laboratory Quiz — Top Ten Most Common Errors



A.3 Information Letter

Attached is a copy of the information letter that I intend to give to students during the first week of classes if the opt-out study is approved

INFORMATION LETTER

Information Letter
Examining the Effect of Enhanced Compilers on Student Productivity

02/01/2017

Research Team

Project Lead: Devon Harker
Department of Computer Science, University of Northern British Columbia
Cell: 250-640-2934
Email: harker@unbc.ca

Supervisor: David Casperson
Assistant Professor and Chair of Computer Science Department,
University of Northern British Columbia
Office: 250-960-6672
Email: casper@unbc.ca

This research is Devon Harker's thesis and is a part of his graduate degree in computer science. The results of this study may be published in scientific journals and may be used to improve the software tools that are used by first year programmers.

Purpose of Project

The software tools used to teach the art of programming to students have not seen much change for many years. This study will help us learn if new software tools will increase student productivity and make it easier for students to complete assignments.

Participation in this research is voluntary and you can withdraw from it at any time with no penalty. You are also free to refuse to answer any questions that make you feel uncomfortable. If you choose to withdraw from the study, any information and data that you have provided will be destroyed unless you consent to the information being preserved and analyzed.

What happens to you in the study? What will be expected of you?

This study will require you to complete the laboratory assignments using software tools that have been installed on the lab computers. Each assignment has been designed so that you can finish it in a single laboratory session and you are not expected nor required to work from home.

There will be two versions of the software tools; the version of the tool that you are using will be randomly decided. Your peers in the laboratory section will be using the same version that you are.

You will be required to answer 3 questionnaires over the course of the semester. The questionnaires are short and can be completed in a few minutes. Questionnaires will be given out at the start of lab sections. You will also have to take a laboratory quiz in the middle of the semester. This quiz is not for marks and will be used as a final test of both your abilities and of the effectiveness of the software tools that you have been using. The quiz will have exam-like conditions (no talking to friends, no looking online for help, etc.)

Risks or benefits to participating in the project

This study is not intended to harm you in any way. If, at any point in the study, you feel uncomfortable or upset and wish to end your participation, please notify the researcher immediately and your wishes will be respected.

By participating in the study, you may have the opportunity to use the latest versions of software tools that are intended to help you write programs. These tools are not publicly available. It is possible that the software you use will affect the difficulty of laboratory assignments and, consequently, the grade you will receive for the laboratory component. Measures are in place to ensure you will not be negatively impacted by the software.

If the software tools prove to be effective at helping novice programmers be productive, future programmers will have an easier learning how to program if the tools become accepted for wide-spread use in teaching programming.

How will your identity be protected?

Beginning next week, you will be given a paper slip with your identifier on it. This study will require you to use the same identifier throughout the entire study. The identifier will be used to link the various pieces of data that will collected to a single person while maintaining your anonymity. You will be required to not share your identifier with anyone as this poses a threat to your anonymity.

The third lab session will require you to send an email to your UNBC email account with a message that contains your identifier. This will allow you to recover your identifier if you ever lose the paper slip.

The data that will be collected will be stored on a database located on the UNBC campus and will not be given to anybody. This database is password-protected and the password is known only to the research team. Your name is required to ensure you are given the same software that your peers in your laboratory session are using. The link between your name and your identifier will be destroyed before the data is analyzed.

Study Results

You will be provided with a summary of the results of the study via your UNBC email address. If you wish to access the entire thesis, it will be stored in the library. The results may also be published in scientific journals.

Who do I contact if I have questions or concerns about the study?

If you have any questions or concerns about the study, the research team would be happy to address them. The contact information of the research team is found at the top of the first page of this form.

Who do I contact if I have complaints about the study?

If you have any concerns or complaints about your rights as a research participant and/or your experiences while participating in this study, contact the UNBC Office of Research at 250-960-6735 or by e-mail at reb@unbc.ca.

Participant Withdrawal

Taking part in this study is entirely up to you. You have the right to refuse to participate in this study. If you decide to take part, you may choose to pull out of the study at any time without giving a reason and without any negative impact on your class standing or anything else.

If you wish to withdraw from the study, you may complete the provided withdrawal form at any time. You will also be given an opportunity to withdraw each time you write a quiz or a questionnaire.

A.4 Withdraw Form

Attached is the form that I intend to use for allowing participants to withdraw from the study.

Examining the Effects of Enhanced Compilers on Student Productivity - Withdraw Form

This form is for participants who no longer wish to participate in the study "Examining the Effect of Enhanced Compilers on Student Productivity".

Withdrawing can be done at any time and with no penalty. When you withdraw, any data collected on your activities will also be destroyed unless you indicate otherwise. Your identifier is required so that the research team knows who to remove from the study.

Identifier ____________________

I am WITHDRAWING from the study ☐

I want all of the data collected on my activities to be PRESERVED for research use ☐

A.5 Programming Experience Pre-assessment

Attached is the entrance quiz that I will use to assess the level of prior programming experience of students in the class.

Programming Experience Pre-assessment

The purpose of the quiz is to evaluate how much experience you have with programming in Java (if any). This quiz is not for marks. Your answers to the quiz questions will allow the professor and teaching assistants to be more effective at teaching the course material to you and future students who take the course. If you don't know the answer to a question, please write "Don't know" (or something equivalent) instead of leaving the answer blank or making a wild guess. You need to write down an anonymous identifier of your choosing below. You will use this identifier for other material in the course so please copy it into your notes for future reference. If you cannot think of an anonymous identifier, you can draw one from a box of pre-made identifiers.

Name ____________________

Identifier ____________________

1. What is a conditional statement? Why are they useful for programmers?

____

____

2. What is the Array data structure used for in Java?

____

____

3. What is the ArrayList datatype used for in Java?

____

____

4. What does the Java statement “x++;” do? Does it do the same thing as “++x;”?

____

____

5. Describe a scenario in a Java program where a do-while loop is a better choice than a while loop.

____

____

6. How can you determine if a program you have created contains any programming mistakes?

____

____

7. Is syntactic sugar mandatory or optional? Explain why.

____

____

8. What is the relationship between a class and an object in Java?

____

____

9. Describe what a pointer is and why programmers would want to use one.

____

____

10. Describe some of the possible values for the following data types in Java:

10a. Integer ____________________

10b. String ____________________

10c. Float ____________________

10d. Double ____________________

10e. Boolean ____________________

A.6 Questionnaires

A.6.1 Questionnaire 1

This Questionnaire is a required component of the "Examining the Effects of Enhanced Compilers on Student Productivity" study that you may have consented to participate in. Your answers will not affect your grade in any way so please be honest with how you feel. If you wish to continue being a part of this research, please fill out the entire questionnaire EXCEPT for the checkboxes that are used to withdraw from the study. If you wish to be removed from this research, please check the first check box, which indicates that you wish to be removed. If you are not participating in the study, please answer questions A and B.

Identifier __________________

I am WITHDRAWING from the study ☐

I want all of the data collected on my activities to be PRESERVED for research use ☐

Question 1: On a scale of 1 through 5, how much prior programming experience have you had coming into this class?

O	O	O	O	O
1:	2:	3:	4:	5:
None	Little	Some	Much	Very Much

Question 2: On a scale of 1 through 5, how much confidence do you have in your ability to solve programming problems?

O	O	O	O	O
1:	2:	3:	4:	5:
None	Little	Some	Much	Very Much

If you are withdrawing or are not participating in the study, please answer the questions below. These questions are not for marks.

Question A: In the Java programming language, the “int” data type is compatbile with both integers and decimal numbers.

☐	☐
True	False

Question B: It is unnecessary to use semicolons to terminate Java statements.

☐	☐
True	False

A.6.2 Questionnaire 2

This Questionnaire is a required component of the "Examining the Effects of Enhanced Compilers on Student Productivity" study that you may have consented to participate in. Your answers will not affect your grade in any way so please be honest with how you feel. If you wish to continue being a part of this research, please fill out the entire questionnaire EXCEPT for the checkboxes that are used to withdraw from the study. If you wish to be removed from this research, please check the first check box, which indicates that you wish to be removed. If you are not participating in the study, please answer questions A and B.

Identifier ____________________

I am WITHDRAWING from the study ☐

I want all of the data collected on my activities to be PRESERVED for research use ☐

Question 1: On a scale of 1 through 5, how much frustration do you experience when fixing errors in your programs?

O	O	O	O	O
1:	2:	3:	4:	5:
None	Little	Some	Much	Very Much

Question 2: On a scale of 1 through 5, how much confidence do you have in your ability to solve programming problems?

O	O	O	O	O
1:	2:	3:	4:	5:
None	Little	Some	Much	Very Much

If you are withdrawing or are not participating in the study, please answer the questions below. These questions are not for marks.

Question A: In the Java programming language, the "boolean" data type is suitable for use in if-statements.

☐ True	☐ False

Question B: The variable name "$123" is legal in Java.

☐ True	☐ False

A.6.3 Questionnaire 3

This Questionnaire is a required component of the "Examining the Effects of Enhanced Compilers on Student Productivity" study that you may have consented to participate in. Your answers will not affect your grade in any way so please be honest with how you feel. If you wish to continue being a part of this research, please fill out the entire questionnaire EXCEPT for the checkboxes that are used to withdraw from the study. If you wish to be removed from this research, please check the first check box, which indicates that you wish to be removed. If you are not participating in the study, please answer questions A and B.

Identifier ____________________

I am WITHDRAWING from the study ☐

I want all of the data collected on my activities to be PRESERVED for research use ☐

Question 1: On a scale of 1 through 5, would you recommend the compiler that you used to other novice programmers?

O	O	O	O	O
1:	2:	3:	4:	5:
Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree

Question 2: On a scale of 1 through 5, how much confidence do you have in your ability to solve programming problems?

O	O	O	O	O
1:	2:	3:	4:	5:
None	Little	Some	Much	Very Much

If you are withdrawing or are not participating in the study, please answer the questions below. These questions are not for marks.

Question A: In the Java programming language, two String variables should be compared with “==” .

☐	☐
True	False

Question B: Every program in Java contains the method “public main static void”.

☐	☐
True	False

A.7 Laboratory Quiz

Attached is the laboratory quiz that was distributed to participants of the study.

CPSC 110 - Laboratory Quiz

March 28, 2017

Introduction

This quiz is a part of the "Examining the Effect of Enhanced Compilers on Student Productivity" study that you may have consented to participate in. This quiz is NOT for marks, but will instead be used to evaluate how quickly and accurately you are able to complete programming problems that are similar to what you have completed previously in the course. The Teaching Assistant for this course will provide feedback on the programs created for this quiz. If you have withdrawn from the study previously, you should still write the quiz anyways; no data will be collected on how you do but the quiz provides a unique opportunity to test your programming skills.

Please read and follow ALL of the following rules:

Rules

- This quiz will have exam-like conditions including, but not limited to: no talking, no cheating or looking at your notes, phones turned off, and unnecessary applications closed.

- Once the quiz has started, you cannot leave the room until you have finished or time runs out, barring exceptional circumstances.

- The invigilator cannot help you fix errors in your programs. However, you may ask for clarification on what a question is asking you to do.

- When you have finished the quiz OR when time runs out, please compile your programs and then print them. Instructions for printing can be found in the glossary, located at the end of the quiz.

- Consult the glossary if you are unsure of how to solve a problem.

- You MUST name your program using the name listed in each quiz question.

- Your programs MUST be saved in a directory named "quiz" (without the double quotes). This directory should be inside the cpsc110 directory that you have been using for your lab assignments. If you are unsure of how to create this directory, consult the quiz invigilator.

Question 1

Write a program named `QuizQuestion1` that displays the following message:

```
Hello World!
```

Figure 1: Example output for QuizQuestion1

Question 2

Write a program named `QuizQuestion2` that asks the user for the length of 3 sides of a right triangle.

- If the 3 side lengths form a legal right triangle, the program should compute the length of the triangle's perimeter as well as its area.
- If the right triangle is illegal, the program should display a message that it is illegal.
- The user MUST be able to enter decimal numbers for side lengths.
- You MUST use either the Math.pow method OR the Math.sqrt method to complete this question (see the glossary).

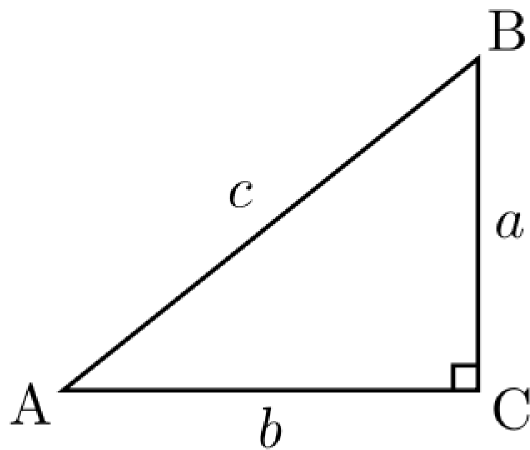


Figure 2: A Right triangle with side lengths of a, b, and c. Credit to: A Malik Pakistan on Wikipedia

```
What is the length of the first side? 3.0
What is the length of the second side? 4.0
What is the length of the third (and longest) side? 5.0


The perimeter of this triangle is: 12.0
The area of this triangle is: 6.0
```

Figure 3: Example output for QuizQuestion2 with legal input

```
What is the length of the first side? 77.5
What is the length of the second side? 0.0
What is the length of the third (and longest) side? -35.0


This triangle is illegal!
```

Figure 4: Example output for QuizQuestion2 with a illegal input

Question 3

Write a program named `QuizQuestion3` that asks the user for 5 integers, which should be stored in an array of integers. Include a swap method that is capable of swapping ANY two elements in the array of integers. This method must be able to take the following as input:

- The index of the first element to swap
- The index of the second element to swap.
- The array that the indices above are for.

Lastly, use a loop to print each element in the array.

Enter 5 integers with spaces between them:

10 11 12 13 14

Enter two indices (start from 0) to swap, with spaces between them:

0 2

The array is now:

12 11 10 13 14

Figure 5: Example output for QuizQuestion3

Bonus Question 1

Write a program named `BonusQuestion1` that asks the user for 5 doubles, which should be stored in an array of doubles. Then, write a method that finds the largest element in the array and prints it. You CANNOT sort the array or force the user to enter it in sorted order.

Enter 5 doubles with spaces between them.

3.0 2.5 7.5 3.5 2.0

The largest element is: 7.5

Figure 6: Example output for BonusQuestion1

Program	Level	Snack
CPSC	Undergrad	Burger
CPSC	Graduate	Spaghetti
PHYS	Undergrad	Pizza
PHYS	Graduate	Sandwich

Figure 7: Snack Matrix

Bonus Question 2

Write a program named `BonusQuestion2` that determines which snack is appropriate for various university students.

```
What is your program? CPSC
What is your level? Graduate


You get Spaghetti!
```

Figure 8: Example output for BonusQuestion2

Glossary

Imports

You may need the following import to complete the questions on this quiz that have user input.

- import java.util.Scanner;

Methods

You may find these methods useful for some of the questions. The return type of a methods is listed first, then the name. The input that a method uses, if any, can be found in the parenthesis after the name. The comment at the end summarizes what the method does.

System Methods

- `void System.out.print(...)` //Prints a message to the screen.
- `void System.out.println(...)` //Prints a message to the screen and inserts a line break after the message.

Math methods

- `double Math.pow(double x, double y)` //Returns the result of x^y.
- `double Math.sqrt(double x)` //Returns the result of $\sqrt{x}$.

String methods

- `boolean String.equals(String b)` //Compares two strings and returns true if they have the same contents.

Formulae

Where a, b, and c are the sides of the triangle.

- Legality of a right triangle: legal if ALL of the following conditions are met.
  - $a > 0$, $b > 0$, $c > 0$
  - $a^2 + b^2 = c^2$
- Area of a right triangle: $area = \frac{a * b}{2}$
- Perimeter of a right triangle : $perimeter = a + b + c$

Boolean expressions

- `a && b` - a and b must both be true for the whole expression to be true.
- `a || b` - The expression is true if a or b are true or if both are true.

Printing Instructions

- First, you must change to the quiz directory (as this is where your programs are located).
  - cd cpsc110/quiz
- Then, use the following set of commands for EACH of your programs. Note: underlined components are placeholders that you need to change.
  - script nameOfTheQuestion.script
  - cat nameOfYourProgram.java
  - javac nameOfYourProgram.java
  - java nameOfYourProgram

- (User input, if any. Use what is shown in the figures by each question)
- exit
- /opt/scriptfix/scriptfix nameOfTheQuestion.script > nameOfTheQuestion.clean
- enscript -2rG -P prn8-457 nameOfTheQuestion.clean

A.8 Enhanced Error Messages Used In Study

This appendix contains an exhaustive list of the error messages that were enhanced by `Decaf`. The format of the error messages has been adjusted to better fit this appendix. The enhanced error messages have been split across two table. In the first table, Table A.30 on page 142 the error messages from `javac` and `Decaf` are compared. Some errors, such as error number 8, vary slightly. The enhanced error messages for each of the variants are included. The second table, Table A.31 on page 144, contains error messages that do not have a `javac` equivalent. These error messages very rarely appear on their own. Instead, they are prefixed to an error message from the first table. The rationale for this process is that the information in the prefixed message may be relevant for solving the error that was reported in the first error message.

Error	*javac*	*Decaf*
4	unclosed string literal	There are mis-matched ”’s on line *number*
5	unclosed character literal	There are mis-matched ”s on line *number*
6	undefined variable	On the specified line there is a mis-spelled or missing variable declaration. Check spelling and that you are not using a variable that is not declared previously.
7	cannot find symbol symbol: variable length	To get the length of a String, use <String name>.length()”
8A	cannot find symbol symbol: variable *name*	The compiler is confused about a variable which is named ”*name*”. If this is supposed to be a method, make sure that there are opening and closing parentheses (something like ”*name*()”). Alternatively, check that ”*name*” has been declared, is in scope, and is spelled correctly.
8B	cannot find symbol symbol: method *name*	The compiler is confused about a method which is named ”*name*”. If this is supposed to be a variable, make sure that there are no parentheses immediately after ”*name*”. Alternatively, check that ”*name*” has been declared and is spelled correctly.

9	')' expected Insert missing ')' where indicated
10	class *name* is public, should be declared in a file named *name*.java Make sure that your class name and file name are the same!
11	variable *name* is already defined in method *name* Variable *name* is already declared, you cannot have multiple identifiers with the same name
12	array required but *type* found An array is required here but a *type* was found
13	invalid method declaration; return type required The method *name* does not have a return type. Make sure the return statement exists and is correct. If the return type should be void, check that you did not forget 'void' as the return type of a method declaration.
14	unreachable statement The statement on the stated line can never be executed. Check that it does not occur after a return, a break, or a continue statement.
15	invalid flag: null It looks like you tried to compile an empty program!

17	′.′ expected import *name* Check import statement on indicated line. Import statements must be of the form ″import packagename.*;″ or ″import packagename.ClassName;″
18	′;′ expected Check for missing semicolon or unnecessary ′(′ or ′)′ on indicated line. If this is for a method declaration, make sure the opening and closing braces that enclose the method′s body are present.
19	′[′ expected [missing on indicated line.
20	′]′ expected] missing on indicated line.
21	variable might not have been initialized variable might not have been initialized. The variable may not always have a value before it is used. Consider initalizing the variable on the line where it is declared (e.g. int x = 0;).
22	not a statement Check indicated line for mis-spellings. If a method is being called, make sure that the number and types of arguments are correct. If the method has no arguments, make sure that empty parenthesis ′()′ appear after method name. Also check that no variable names start with numbers or other disallowed characters. Check that you did not use == where you meant to use = . Check that you did not use + = instead of += . Also check for a stray semicolon

23	illegal character Check your ′ and ″. If you copied and pasted code from a word processor, the web, or another source you may have to delete and replace them with characters typed in this editor.
24	illegal start of expression Check the following: Did you type something like x + = 1 instead of x+=1? If in a switch statement, make sure you type ′case something:′ instead of ′case: something′ Make sure you are not writing a method inside another method. Make sure you are not declaring a static variable inside of a method.
25	invalid type expression Check for a missing ; on the indicated line.
27	<identifier> expected Check three things: Are you trying to use a variable before it has been declared? For example, did you write ″x = 3;″ rather than ″int x = 3;″? If so, then declare the variable first. If this is a statement and it is outside of a method, try moving it inside of a method. If this is a method declaration, make sure you are not using ′void static′. A static and void method must be declared ′static void′.
28	method *name* not found in class *name* undefined (missing) method on line indicated. Did you write something like MyClass y = MyClass() instead of MyClass y = new MyClass ?

29A	return required Ensure the method ending on this line (with this '}') has a return statement, which returns a type indicated in the method's declaration.
29B	missing return statement Ensure the method ending on this line (with this '}') has a return statement, which returns a type indicated in the method's declaration.
30	non-static variable *name* cannot be referenced from a static context Are you trying to use a variable declared outside of a method? Or perhaps you are using a method without trying to apply it to an object? In both cases, you may be able to fix it by writing "static" before the declaration of the variable or method. Also make sure you are not writing a method inside another method.
31	bad operand type String for unary operator '+' The + operator can only be used between two Strings. Most likely try eliminating the +, otherwise perhaps you forgot a String variable in the expression.

32	error: incompatible types: possible lossy conversion from *type* to *type* Look for a statement such as i = d; where i is an int, and d is a double. If this is intended, you need to cast the second type to the first. In this case, the statement that avoids the error is i = (int)d; If this is a method call, make sure that you use the correct types; For example, methods that expect an integer will not work if given a double. If you are trying to look at an element at some index in an array, make sure your index is an int.
33	reached end of file while parsing Most likely you have too few closing braces '}'.
34	has no definition of serialVersionUID The reason for this error is complex. To avoid it, enter the following line inside the class where the error is occurring: public static final long serialVersionUID = 1L;
35	incompatible types: *type* cannot be converted to *type* Check the datatype of both sides of the expression with "=" , they should be of the same datatype. Also, check if you are using "=" where there should be "==" If this is an argument for a method, check if the method expects an array or a regular variable.

36	'else' without 'if' Check the placement of the branches; else-if branches can only go right after an if branch or another else-if branch. Else branches can only go right after all of the related if and else-if branches. Check to make sure the opening and closing braces (the '{' and '}') of all the branches are in the right spot. Also check for misplaced semicolons in all branches, such as "if(x); {...}"
39	cannot find symbol symbol: class string If "string" refers to a datatype, capitalize the "s"!
40	package system does not exist Capitalise "system" so it reads "System"!
50	duplicate case label There are two or more case statements in this switch block that have the same label (e.g. the "2" in "case 2:"). Look for the duplicate and either remove it, rename it, or combine its body with other cases that have the same label.
9000	no suitable method found for *name*(*types*) The compiler cannot determine which method you were trying to use, probably due to an error with the arguments in the parentheses. Try changing the arguments when you call the method to match one of the methods shown above.

9001	cannot find symbol symbol: method nextint Are you trying to read an integer with a Scanner? Use nextInt(). If not, double check your spelling and make sure everything has been declared.
9002	cannot find symbol symbol: method nextline *OR* method nextstring *OR* method nextString Are you trying to read a String with a Scanner? Use nextLine(). If not, double check your spelling and also make sure everything is declared.
9003	cannot find symbol symbol: variable nextint *OR* variable nextInt Are you trying to read an integer with a Scanner? You may be missing the brackets that tell the compiler that nextInt is a method, try using nextInt().
9004	cannot find symbol symbol: variable nextline *OR* variable nextLine *OR* variable nextstring *OR* variable nextString Are you trying to read a String with a Scanner? You may be missing the brackets that tell the compiler that nextLine is a method, try using nextLine().
9005	bad operand types for binary operator ˆ Are you trying to apply exponents to numbers? You need to use the pow method of the Math class. Try ”Math.pow(base, exponent)” where the base and the exponent are number literals, variables, or expressions.

9006	missing method body, or declare abstract If there is a semicolon near your method declaration, remove it. Otherwise check to make sure there are opening and closing braces after the method header.
9007	'.class' expected If you are trying to call a method while using variables as arguments, do not include the types of the variables in the method call, as the type should already be defined in the method declaration.
9008	bad operand types for binary operator '&&' If you are trying to do AND as part of a condition, double check that both sides of the && are booleans. Also make sure you are using == instead of = when checking for equality.
9009	bad operand types for binary operator '||' If you are trying to do OR as part of a condition, double check that both sides of the || are booleans. Also make sure you are using == instead of = when checking for equality.

9010A	method *name* in class *name* cannot be applied to given types. required: *types*. found: no arguments It looks like you are trying to call a method named "*name*" with incorrect arguments. This method was expecting the following set of arguments: *types*. However, nothing was found in the parentheses when you called the method. Double check that you are calling the correct method. Then double check that you have all of the values or variables that the method needs to use. Lastly make sure that the order of the arguments is in the order that is defined in the method declaration.
9010B	method *name* in class *name* cannot be applied to given types required: no arguments found: *types* It looks like you are trying to call a method named "*name*" with incorrect arguments. The compiler was expecting to find nothing in the parentheses when you called the method. However, the compiler found the following arguments instead: *types*. Double check that you are calling the correct method. Then double check that you have all of the values or variables that the method needs to use. Lastly make sure that the order of the arguments is in the order that is defined in the method declaration.

9010C	method *name* in class *name* cannot be applied to given types required: *types* found: *types* It looks like you are trying to call a method named "*name*" with incorrect arguments. This method was expecting the following set of arguments: *types*. However, the compiler found the following arguments instead: *types*. Double check that you are calling the correct method. Then double check that you have all of the values or variables that the method needs to use. Lastly make sure that the order of the arguments is in the order that is defined in the method declaration.

Table A.30: Comparison of `javac`'s and `Decaf`'s Error Messages

Error	*Decaf*
2	Your program has unmatched braces (there is/are *number* more opening braces '{' than closing braces '}'). This may or may not be related to the current error. Add matching braces where necessary. If the braces are appropriately matched, check for other errors such as unclosed double-quotes ("). There may be a missing '(' on line *number* Alternatively check for missing " or mis-spellings on this line.
3	Your program has unmatched braces (there is/are *number* less opening braces '{' than closing braces '}'). This may or may not be related to the current error. Add matching braces where necessary. If the braces are appropriately matched, check for other errors such as unclosed double-quotes ("). There may be a missing ')' on line *number* Alternatively check for missing " or mis-spellings on this line.
37	Your program has unmatched parentheses (there is/are *number* less opening parentheses '(' than closing parentheses ')'). This may or may not be related to the current error. Add matching parentheses where necessary. If the parentheses are appropriately matched, check for other errors such as unclosed double-quotes (").
38	Your program has unmatched parentheses (there is/are *number* more opening parentheses '(' than closing parentheses ')'). This may or may not be related to the current error. Add matching parentheses where necessary. If the parentheses are appropriately matched, check for other errors such as unclosed double-quotes (").

41	The program has an odd number (*number*) of ”double-quotations”. This means that one of them may be unclosed and this could be causing the error.
500	While not technically an error, something unusual was detected. The while loop on line *number* has a semicolon immediately after it and it is not part of a do-while loop. Is this intended?
501	While not technically an error, something unusual was detected. The for loop on line *number* has a semicolon immediately after it. Is this intended?

Table A.31: `Decaf` Exclusive Error Messages