

Techniques for Enhancing Compiler Error Messages

A Dissertation

Presented in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Sanaa M. Algaraibeh

Approved by:

Major Professor: Terence Soule, Ph.D.

Co-Major Professor: Clinton Jeffery, Ph.D.

Committee Members: Jim Alves-Foss, Ph.D.; Tonia Dousay, Ph.D.

Department Administrator: Terence Soule, Ph.D.

May 2023

Abstract

Bottom-up parsing technology advanced and efficiently automated static analysis of source code but raised the challenge of maintaining understandable communication between compilers and humans. Reporting errors in the source code in a humanly understandable language is essential for the efficiency of the software development process, especially for students learning programming. This research improves compiler error messages for students in introductory programming courses at the college level. This study of compiler error messages is written from the compiler writer's perspective. Analysis of hundreds of erroneous programs in different parsing states led to designing 3-phase parsing techniques that overcome some of the limitations of LR parsers in reporting friendly error messages. 3-phase parsing prioritizes the parsing of the large code components over diving into all the details. The first phase parses the functional structures and ignores errors in the syntax of the smaller constructions. The second phase parses the control structures and ignores errors in the expressions and other statements. The third phase parses the expressions and statements excluded from phase two. The design gives more control over when and from which grammar rules to report errors first. The design minimizes the number of states in the parser automaton since each phase parses a subset of the language grammar.

We evaluated the new design with a human-subject control experiment. The experiment compares the quality of syntax error messages of an educationally customized compiler EduCC with error messages produced by GNU GCC and Microsoft Visual C++. EduCC implements the 3-phase parsing techniques for a subset of C++ language. The participants were 53 Computer Science and Engineering students at the New Mexico Institute of Mining and Technology. In a within-group experiment design, the participants had to find errors and fix erroneous C++ programs. The experiment shows that 3-phase parsing techniques improved the quality of syntax error messages

Acknowledgments

I am incredibly grateful to my advisor, Dr. Clinton Jeffery. His kindness, expertise, and intellectual legacy guided me and will always do. Likewise, I thank my advisor Dr. Terence Soule for his guidance, instructions, and mentoring. Without their support, this work would not have been possible.

I thank my committee member and my educational mentor, Dr. Tonia Dousay. She is my inspiration. In addition, I thank my committee member, Dr. Jim Alves-Foss, for his support and encouragement.

I thank my friends and fellows, Amruta Kale and Nael Radwan, for their support and help. Finally, I would like to thank Amy Knowles and the other professors at the New Mexico Institute of Mining and Technology who encouraged their students to participate in the experiment of this work.

Dedication

I dedicate this work to my mother, who is with Alzheimer and still loves and asks about me.

To my sisters, especially Fatima and Alia, whose endless support and encouragement make this work possible.

Finally, I dedicate this work to my brothers, nieces, and nephews, who are proud of what I accomplished before I did it.

Table of Contents

Abstract	ii
Acknowledgments	iii
Dedication	iv
Table of Contents	v
List of Figures	viii
List of Tables	x
Statement of Contribution	xi
Biography	xii
Chapter 1: Introduction	1
1.1 Introduction	1
1.2 Dissertation Objective	1
1.3 Research Question	2
1.4 Methodology	2
1.5 Terminology	2
Chapter 2: Background and Related Works	4
2.1 Compiler error messages are often unhelpful	4
2.2 How should compiler report syntax errors	4
2.3 Can we overcome some limitations of the parser in generating better error messages?	6
2.4 Methodology and measurement of effectiveness of compiler error messages	7
Chapter 3: Integrated Learning Development Environment for Learning and Teaching C/C++ Language to Novice Programmers	10
3.1 Introduction	10
3.2 Challenges facing novice programmers	11
3.3 ILDE	13
3.5 Conclusion	18
Chapter 4: Analysis of Syntax Error Messages from the Learner’s Perspective	19

4.1 Analysis of common errors in the syntax of the function body.....	19
4.2 Analysis of common errors in the syntax of the if statement	25
4.3 Analysis of common errors in the syntax of the for statement.....	29
4.4 Conclusion.....	33
Chapter 5: Engineering a Compiler for Better Error Messages.....	34
5.1 Introduction	34
5.2 Compiler.....	34
5.3 Parser.....	35
5.4 Error detection.....	38
5.5 New solution: 3-phase parsing techniques.	45
5.6 Implementation of the 3-phase parsing techniques in an Educationally Customized Compiler:	47
5.7 Sample error messages generated by EduCC	55
5.8 Conclusion.....	64
Chapter 6: Evaluation of Error Message Quality Enabled by 3-Phase Parsing Techniques	65
6.1 Introduction	65
6.2 Methodology	65
6.3 Results	73
6.4 Limitations.....	76
6.5 Conclusion.....	77
Chapter 7: Conclusion and Future Work.....	78
References	81
Appendices	84
Appendix A: Study Materials for Experimental Approach to Evaluate Messages Enabled by 3-Phase Parsing Techniques.....	84
A.1 Invitation Letter.....	84
A.2 Consent Form	85

A.3 Tasks.....	86
A.4 T-Test for (RQ1: Is there a significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in finding syntax errors?)	92
A.5 T-Test for (RQ2: Is there a significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in fixing syntax errors?)	93
A.6 T-Test (6.3.1 RQ2: Is there a significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in the time-to-find and -fix?).....	94
Appendix B : Source code of the EduCC	95
B.1 meta.err for parser 1	95
B.2 meta.err for parser 2.....	96
B.3 yyerror.c for parser 1	97
B.4 berror.c for parser 2	99
B.5 main.c.....	101

List of Figures

Figure 3.1 Example of misconception: “uninitialized memory allocation.”	13
Figure 3.2 Example of memory visualization of <code>for(i=1; i<=7; i++) weekly_product</code> <code>+=daily product;</code>	16
Figure 3.3 Example of CPU visualization.....	17
Figure 4.1 Some C++ programs with common syntax errors of novice programmers: unbalanced curly brackets.....	20
Figure 4.2 Good quality syntax error messages reported by GCC and VC++on compilation of prog1.cpp in Figure 4.1	20
Figure 4.3 Bad quality syntax error messages reported by GCC and VC++on compilation of prog2.cpp in Figure 4.1	21
Figure 4.4 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog3.cpp in Figure 4.1	22
Figure 4.5 Bad quality syntax error messages reported by GCC and VC++on compilation of prog4.cpp in Figure 4.1	22
Figure 4.6 The C++ programs From Figure 4.1 with drawings that clarify how a parser recognizes the function body boundaries.....	24
Figure 4.7 C++ programs with common syntax errors of novice programmers if statement header. ...	25
Figure 4.8 Bad quality syntax error messages reported by GCC and VC++on compilation of prog5.cpp in Figure 4.7	26
Figure 4.9 Bad quality syntax error messages reported by GCC and VC++on compilation of prog6.cpp in Figure 4.7	26
Figure 4.10 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog7.cpp in Figure 4.7	27
Figure 4.11 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog8.cpp in Figure 4.7	28
Figure 4.12 C++ programs with common syntax errors of novice programmers, for statement header.	29
Figure 4.13 Bad quality syntax error messages reported by GCC and VC++on compilation of prog9.cpp in Figure 4.12	30
Figure 4.14 Bad quality syntax error messages reported by GCC and VC++on compilation of prog9.cpp in Figure 4.12	31

Figure 4.15 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog11.cpp in Figure 4. 12	31
Figure 4.16 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog12.cpp in Figure 4.12	32
Figure 5.1 Compiler architecture.....	34
Figure 5.2 Model of an LR parser [47].....	37
Figure 5.3 State 0 of the automaton of the grammar in Figure 5.1.....	42
Figure 5.4 State 2 of the automaton of the grammar in Figure 5.1.....	42
Figure 5.5 State 34 of the automaton of the grammar in Figure 5.1.....	44
Figure 6.1 Example of page three/four of the Qualtrics web page for the experiment.	71
Figure 6.2 Example of part two of the Qualtrics web page for the experiment.....	72
Figure 6.3 Participants' answers for the question "Do the compiler error messages correctly give the location (line) of the actual error?"	74
Figure 6.4 Participants' answers for the question " Do the compiler error messages describe what is the actual error?"	75
Figure 6.5 Participants' answers for the question "Do the compiler error messages suggest how to fix the error?".....	76

List of Tables

Table 6.1 Age and gender of the participants.....	66
Table 6.2 The participants programming experiences.....	67
Table 6.3 The groups, programs, compiler, type of error, and number of respondents.	70
Table 6.4 Success rate of answering the question “what is the error in the program?”	74
Table 6.5 Success rate of answering the question “in which line is the error?”	74
Table 6.6 Success rate of answering the question “what is the cause of the error?”	74
Table 6.7 Success rate of answering the question “how to fix the error?”	75

Statement of Contribution

The work described in Chapter 3 was primarily the work and authorship of Sanaa Algaraibeh. Dr. Clinton Jeffery and Dr. Dousay acting in their role as Major Professor and Committee member gave input on advising and reviewing the draft versions of the chapter.

Biography

Sanaa Algaraibeh is an instructor at the New Mexico Institute of Mining and Technology. She worked in academia for 14+ years as a lecturer, trainer, team leader, instructional designer, and CS department chair at universities in Jordan and Saudi Arabia. She teaches Internet and Web Programming, Object-Oriented Programming, Python for Data Science, and Introduction to Programming. Her area of scholarship is computer science education. She is interested in developing computational solutions integrated with modern pedagogy.

Chapter 1: Introduction

*“To err is human;
to fix is divine.”*

1.1 Introduction

Programming is a required competency in the journey of computer science students and for students in related fields. Some students start learning to program in high school or earlier. They need to have skills in converting real-life problems into computer solutions. They need to understand the notional machine, the syntax, and semantics of one or more programming languages, develop problem-solving skills, design solutions, write code, and debug their code. Many college CS curricula begin with C++ or Java in their first courses for majors, and with good reason. But the relative difficulty of these languages makes the need for an educationally customized compiler greater than it is in "easier" first languages. The C++ language and its family of languages, like Java, are designed to be strict and precise and are meant for expert programmers, not for novices.

Background and related works are presented in Chapter Two. Chapter Three presents our educational development environment for C/C++. Chapter Four gives an analysis of the quality of syntax error messages. Chapter Five presents a new solution, a 3-phase parsing technique to help generate better error messages. Chapter Six presents an experimental design that evaluates the quality of syntax error messages generated by a compiler implementing the proposed technique. Finally, Chapter Seven summarizes the results of the dissertation.

1.2 Dissertation Objective

The objective of this dissertation is to enhance compiler error messages to be more helpful to learners. To do this, the design of a 3-phase parsing technique is presented that overcomes some the limitations of how parsers report errors.

The goals to achieve the dissertation objective are:

- Investigate the challenges that face novice programmers.
- Design a solution to the limitation of parsers in reporting syntax error messages.

- Evaluate the quality of syntax error messages of an educationally customized compiler compared to mainstream compilers by testing whether the compiler messages help learners find and fix syntax errors.

1.3 Research Question

Can modified parsing techniques help in generating better syntax error messages?

1.4 Methodology

This study begins by reviewing the common errors of novice programmers in the literature for C++ language and languages from the same family, such as Java. Next, it analyzes how popular compilers used in introductory courses report these common errors. Then, it presents the design of a 3-phase parsing technique that can be applied to many programming languages. The prototype for a compiler that uses the proposed 3-phase parsing technique for C++ is presented. Finally, an experiment is conducted to test whether the proposed parsing technique enhances the quality of syntax error messages.

1.5 Terminology

Syntax error: a parsing failure during the validation of code against the rules of the language grammar. The compiler judges whether the source code follows the rules or not; if not, it has a syntax error. So, if a program has an error in the structure, such as missing or extra token(s), or if the order of tokens is incorrect, the source code has one or more syntax errors. If the source code has an error, the compiler stops the compilation process and tells the user about the errors.

Syntax error message: a message from a compiler to a user reporting a syntax error.

Quality of syntax error message: syntax error message can be classified as bad, good, or innovative. The following definitions are inspired by the software engineering definition of quality [1].

Bad quality: a syntax error message does not deliver what the average programmer expects the message to have. So, if the message is incorrect or not understandable, it is a bad message. This level of quality confuses and misleads programmers, especially novices.

Good quality: the syntax error message achieves the basic requirements of error messages. It is sufficient to help the programmer in finding and fixing a syntax error.

Innovative quality: the syntax error message exceeds good quality by incorporating one or more novel values or features. For example, it may have tailored error messages to different programmer levels, such as novice or expert.

Core elements from proposal of this dissertation were presented and published at the ACM Conference on International Computing Education Research (ICER) doctoral consortium in August 2022, Lugano, Switzerland. Also, it received feedback from the ICER community [2]. Some of their research is discussed in Section 2.4 of the literature chapter.

Chapter 2: Background and Related Works

A compiler is a communication tool; one of its primary roles is to detect errors and report those errors in understandable messages. Today's state-of-the-art compilers are efficient but still give poor error messages for common syntax errors. In general, human-computer interaction research recommends that a software system should help users recognize, diagnose, and recover from errors. Molich and Nielsen assert that "Any system designed for people to use should be easy to learn and remember, effective, and pleasant to use" [3]. For programmers, especially novices, it is important that the compiler provides helpful error messages to enable them to correct their erroneous programs.

Mainstream compilers sometimes give good error messages for common syntax errors, but sometimes they behave in strange ways, even for the same type of errors. Section 4.2 gives an in-depth example of this type of troublesome behavior. These messages often confuse learners and make fixing errors harder.

2.1 Compiler error messages are often unhelpful

In 2014, a study conducted at Google found that expert developers spent significant time and effort correcting common compiling errors. The study analyzed 26.6 million builds of software written in C++ and Java languages and involved 18,000 developers over nine months. The results showed that the average number of builds for C++ developers is 10.1 times a day and for Java developers is 6.98 times a day, and 37.4% of C++ builds, and 29.7% of Java builds fail [4].

In 2019, a survey conducted by Becker et al. showed that both current and old research describes compiler error messages as not understandable, useless, inadequate, frustrating, cryptic, and confusing, undecipherable, intimidating, still very obviously less helpful than they could be, and a barrier to progress [5].

2.2 How should compiler report syntax errors

The research community continues to study and discuss the quality of compiler error messages. In 1982, Shneiderman studied the impact of error messages on users [6]. They conducted several controlled experiments. In one experiment, they modified a Cobol compiler to generate more specific error messages, and they asked groups of students to repair erroneous programs using the modified compiler and the regular one. Then they compared the results. They found that the modified compiler increased the repair scores by 28 percent. Shneiderman recommended that the error message should 1) have a positive tone, 2) tell the user what must be done. 3) use the user's terms, 4) avoid negative terms such as "illegal", "invalid", "error", or "incorrect". 5) avoid obscure terms such as "syntax error." 6) be comprehensible.

In 1983, Brown experimented with testing the quality of error messages [7]. He analyzed the error messages of fifteen Pascal compilers for a group of simple and common erroneous programs. He found that most compilers in the experiment ranged from "barely accepted" to "laughable." Brown recommended that error messages be friendly, give an informative message, and provide help in correcting the error. He also suggested that the system should show the correct possibilities. Compiler writers should avoid compiler terminologies in the messages such as "lexical error" and "syntax error". Brown focused on how helpful the error message would be if it pointed to the offending area of the source code.

In 1998, Lewis and Mulley analyzed the usefulness of error and warning messages for different user levels, received user feedback for years, and improved the compiler error messages accordingly [8]. The compiler was locally built for Modula-2 and used by students over several years in their department. Although they didn't measure the effectiveness of improved compiler warning and error messages, they developed a group of merits of the useful compiler warning and error messages: 1) The compiler error message needed to be helpful. 2) The compiler message hints at how to fix the problem. 3) The compiler should take into consideration different user levels, for example, a user can use -students flag for more detailed error checks. The compiler should also provide extra warnings appropriate for first-year students. These warnings include checking for identifiers that look like keywords, same name variables in different visible scopes, variables and parameters declared but never used, and variables used before being initialized. 4) The compiler in some cases provides a description of what the compiler believes it has seen.

In 2010, Traver proposed a set of principles that should guide compiler error message design [9]. These principles are clarity and brevity, specificity, context insensitivity, locality, proper phrasing, consistency, suitable visual design, and extensible help. In addition, Traver discussed the problem of the context-sensitivity of errors, in which the compiler gives different error messages for the same error.

In 2011, Marceau et al. investigated the effectiveness of error messages for the DrScheme environment [10]. DrScheme is designed for students to teach and learn the Scheme language and to deliver suitable multi-level error messages. To find the shortcomings in the DrScheme's error messages, they looked at how the students edited different errors, interviewed, and quizzed them. They suggested that the metrics of a rubric that measures the effectiveness of good error messages are: 1) students can read the message, 2) students can understand the message, 3) students can formulate a response for the error. 4) students can fix the error. Also, they found that using technical vocabularies in the error message makes it unclear instead of helpful. Encountering programming

terminologies that students have not studied before confuses them, such as “function body” for beginner users. Marceau et al. found that students prefer a hint on how to fix the error over a highlight of the error area in the source code.

In 2018, Barik et al. postulated that the quality of compiler error messages improved if they contained explanations; the compiler error message should apply explanation theories, such as Toulmin’s model of argument [11]. In their study, they mapped the compiler error messages to the reasoning model for Toulmin. The simple components of argument theory are the claim, the ground (evidence), and the warrant (the bridge between the claim and the ground). And it may have the extended component: backing. The study conducted three experiments. The first one asked professional developers from software companies to prefer selected error messages from OpenJDK, and Jikes, where some messages follow the reasoning model, and some do not. The other experiments compare the structure and content in Stack Overflow with compiler error messages. The authors recommended that the designer and developers of compilers should distinguish fixes from explanations and apply argument structure and content to the design and evaluation of error messages. The study showed that developers prefer error messages that use the explanation model over the error messages that do not use the explanation model, but also, they prefer the elaboration over the explanation. The authors recommended that developers may selectively need more or less help in comprehending the problem, and designers and developers of compilers should support mechanisms to progressively elaborate error messages. For example, some static analysis tools, such as Error-Prone, implement such an approach; the tool initially provides a simple argument for the error messages but also enables additional backing through a supporting link.

Some studies paid attention to the readability of error messages. For example, Barik et al. found that programmers spent up to 25% of their task time reading error messages[12]. The study used eye-tracking research techniques with programmers. Another study by Becker et al. found that the key factors that affect the readability of error messages are 1) length of the message, 2) message tone, and 3) use of jargon [13].

2.3 Can we overcome some limitations of the parser in generating better error messages?

Hristova et al. developed a preprocessor tool called Espresso to detect common Java errors in the novice Java codes and report friendly and better error messages with hints on how to fix them [14]. The implementation of the preprocessor was written in C++. They do not mention or discuss the design of how they parse and detect these common errors in the source code. However, their paper was well known for identifying and categorizing the common Java errors of novices using the survey method.

Kohn developed a parser for Python programs capable of recognizing a group of error patterns [15]. He was addressing the problem of Python error messages for high school students. The group of error patterns is identified as part of his research. He studied the issue of misconceptions, and he found part of the problem is that students project the mathematical mental model when writing code. So, they write incorrect expressions in Python programs. He implemented the parser as part of a successful educational environment (TigerJython) that delivers error messages in the German language. The limitation of their parser is that not all novice errors derive from incorrect mathematical mental models.

Jeffery's approach, which is applicable for different programming languages, is an open-source code tool called Merr [16]. Merr takes additional information from the parsers generated by the Yacc and Bison family and passes them to the compiler error message system. More accurately, Merr automatically generates the code of the error reporting function in the compiler (e.g. `yyerror()`) from a set of example errors and messages. In addition, it provides the state that the error occurs on and the erroneous token. The downside of the Merr approach is that the compiler writer must come up with erroneous fragments to associate messages with the states of the parser where errors can occur.

Pottier argued that an LR parser could generate good diagnostic messages using Jeffery's approach [17]. He elaborates on it by designing an algorithm that automatically generates erroneous fragments. Also, he proposed three features of a good diagnostic message that is mapped from the erroneous fragments by the Merr tool. Pottier proposed: "correctness (i.e., every sentence is erroneous), irredundancy (i.e., no two sentences lead to the same state), and completeness (i.e., some sentence reaches every state where an error can occur)." [17]

2.4 Methodology and measurement of effectiveness of compiler error messages

As the experimental approach is established in human-computer interaction and software engineering research, this section reviews relevant empirical studies in enhancing compiler error messages.

Marcus et al., in their study "Measuring the Effectiveness of Error Messages Designed for Novice Programmers," suggested a rubric to measure the edits students made to correct erroneous programs using the enhanced error messages [10]. They collected data from 53 students in an introductory programming course. They used the DrRocket programming environment to collect copies of students' edits during the course lab sessions. DrRocket has been part of the Scheme/Rocket ongoing project for years and aims to enhance compiler error messages for the Scheme language. The rubric categories are failure-on-read, failure-on-understand, failure-on-formulate, and fixed-error. Then they combined the failure-to-read and failure-to-understand. The rubric components are:

“1) delete the problematic code wholesale, 2) unrelated to the error message and does not help, 3) unrelated to the error message, but it correctly addresses a different error or makes progress in some other way, 4) evidence that the student has understood the error message (through perhaps not wholly) and is trying to take an appropriate action (though perhaps not well) , 5) fixes the proximate error (though other cringing errors might remain).”

They reported at the end of the study that the analysis of students' edits provided them with insight into students' performance with the DrRocket environment and in the course. However, they reported that further data analysis is required to develop conceptual problems underlying the students' edits.

Becker, for his Ph.D., studied whether the enhanced compiler error messages are effective in helping students learn to program[18]. They experimented with measuring the effectiveness of modified compiler error messages. They used metrics to compare control and intervention groups: "the total number of errors in each group, the number of errors per student, the number of repeated errors, and the repeated error density."

They recruited two cohorts of 100 students each with one academic year between them. First, they collected data from the control group that used the regular Java SE 7's compiler. Then, the following year, they collected data from the intervention group that used 30 modified messages along with the regular Java SE 7's messages. The collected data were compiler ID, line of code and class of error, compiler error messages, enhanced compiler error messages (for the intervention group), and date/time. For the analysis, they used Wilcoxon signed-rank and Mann-Whitney U tests. For the results, they reported that the intervention group experienced reductions in the number of overall errors, errors per student, and several repeated error metrics. Also, they claimed that the results are generalizable to other programming languages, students, and institutions.

Kohn, in his Ph.D. dissertation, to answer the research question: "do the improved error messages help in learning?" conducted a survey [15]. He developed a parser to enhance Python compiler error messages and integrated it with the known TigerJython IDE that he created for learning and teaching a subset of Python to high schools in Switzerland. He recruited 82 students from three different high schools from teachers in an education program at a university. The questions were rating questions on a scale from 1 (not helpful, do not agree) to 5 (very helpful, strongly agree). The questions were about learning materials, and three were about error messages. The questions about error messages were "Of how much help are the error messages shown in the environment for your learning? How strongly do you agree to the following statements? 1)The German translations of error messages are helpful. 2) I do not read the error messages." They used descriptive analysis for the survey. They calculated the

average and the standard deviation of the respondents' answers. Finally, they reported that they did not find any correlation between error messages and learning. However, he found that students highly appreciated translating the error messages into their native language.

Barik, in his Ph.D. dissertation, conducted studies with others on the impact of compiler error messages on users [19]. The first experiment was to answer the research question: “Are compiler errors presented as explanations helpful to developers?”. They recruited 68 expert full-time software developers at Microsoft. A questionnaire presented Java erroneous snippets with accompanying error messages from both Jikes and OpenJDK compilers and asked the participants which one they preferred. They used five erroneous Java programs that were seeded with common errors. To infer which compiler is better, they used the Chi-squared test. The second experiment was eye-tracking to answer research questions “1) how effective and efficient are developers at resolving error messages? 2) do developers read compiler error messages? 3) are compiler errors difficult to resolve because of the error message?”. They recruited 56 students from undergraduate and graduate software engineering courses at North Carolina State University. They observed the participants as they resolved common errors. For the analysis, for question one, they calculated efficiency from two time-derived metrics: time to complete a task and participant effort. Then they performed a two-tailed t-test between task times, excluding timeouts, under correct and incorrect solution conditions to gauge response time effort. For question two, the measurement was for the meantime of fixations of reading; they computed across participants the percentage of fixations for the areas of interest in the task. Then, they compare the fixation time spent on the source code to the fixation time spent on the error messages. For question three, they used the eye-tracking measure of revisits. That is, leaving an area of interest and then returning to it as a measure of reading difficulty. Finally, they computed a nominal logistic model between correctness and revisits to error messages. The model’s output is a probability of correctness against the number of visits over a distribution of tasks. To evaluate the model, they computed Nagelkerke’s coefficient of determination, R^2 , and a likelihood-ratio Chi-square test (G^2).

Chapter 3: Integrated Learning Development Environment for Learning and Teaching C/C++ Language to Novice Programmers

This chapter is adapted from:

S. M. Algaraibeh, T. A. Dousay and C. L. Jeffery, “Integrated Learning Development Environment for Learning and Teaching C/C++ Language to Novice Programmers,” a work-in-progress paper that appeared in *2020 IEEE Frontiers in Education Conference (FIE)*, 2020, pp. 1-5, Doi: 10.1109/FIE44824.2020.9273887.

This chapter presents an Integrated Learning Development Environment (ILDE) that integrates technologies with pedagogies for first-year students learning to program. It is a proposed design, within which the core contribution of this dissertation constitutes one component. Novice programmers must overcome misconceptions, debugging, and problem-solving. ILDE employs multimedia learning content, formative feedback, a customized compiler, and visualization using modern pedagogical and cognitive psychology practices. Visualization and multimedia illustrate what happens inside the computer as the program is running. Enhanced compiler messages with graphical representation reduce the difficulty of compilation errors.

3.1 Introduction

A student in the first-year computer science course finds themselves like Alice in Wonderland. There are many new mysterious concepts, and they must acquire a large amount of knowledge and many new skills. Programming is a primary competency and a prerequisite for almost all CS courses. It is the basic building block in their journey. Many CS educators express that their students lack programming skills, even after prior programming courses [20]. Much like Wonderland, learning how to program feels like a multidimensional journey filled with different perspectives and tasks.

ILDE integrates technologies with pedagogies. The design of ILDE is built upon Cognitive Load Theory [21],[22], Kolb’s Experiential Learning Theory [23], Constructivism principles [24], Cognitive Theory of Multimedia Learning [25], software visualization technology, and an educationally customized compiler. The learning content addresses core programming competencies taught in computer science education for first- and second-year university students. ILDE merges related course topics: Computational Thinking and Problem Solving, Programming Languages, Computer Operating Systems, System Software, and Discrete Mathematics. Further, ILDE uses problems and projects from real-life contexts to support meaningful learning. The learning activities target complex programming skills developed from related subskills.

The ILDE model is based upon two elements: 1) ILDE's Multimedia Learning Content deploys visualization of computation to enhance novices' performance and 2) ILDE's Informative Feedback utilizes a customized compiler that enhances novices' performance. For both hypotheses, novice performance will be measured by observing whether using ILDE changes course success rates (defined as a grade of C or above) in a statistically measurable manner. This dissertation contributes towards the evaluation of Hypothesis 2.

3.2 Challenges facing novice programmers

Learning to program is difficult. Required competencies include comprehension of programming concepts and ability to write code. Students must learn to convert real-world problems into computer solutions. Fuller et al.'s [26] learning taxonomy sums up the objectives of introductory programming courses as follows: recognize, understand, analyze, and evaluate the programming concepts; apply these concepts by writing code to solve a problem similar to the problems already learned; create code to solve new problems. The learner must also understand the machine operations that a programming language expresses [27]. Why is it so difficult to teach or learn to program? Novices find programming challenging due to fundamental misconceptions, as well as their lack of debugging and problem-solving skills [14, 28, 29].

Misconceptions

Coding misconceptions arise from a lack of knowledge or a false assumption. For example, students may misunderstand the mechanism of a loop, or the relationship between language constructs and underlying memory usage [14, 15, 27, 29, 31]. ILDE has an innovative method for fixing misconceptions. Consider uninitialized memory allocation, a common error for novices: it arises due to a misconception about the relation between memory and language elements. Figure 1.1 shows C++ and Java code that uses an object without initialization. When compiling `mmr4.cpp`, `g++` doesn't report an error; when it is run, a "Segmentation fault" message appears. This message is mysterious to novices and doesn't help them understand or fix their problem. Encouraging novices to take the compiler's warnings seriously can help. `g++ -Wall` reports " 'generator' may be used uninitialized in this function". The Java compiler produces a "variable number might not have been initialized" error. Microsoft Visual Studio and BlueJ IDE show similar error messages for this type of error.

To address this misconception, ILDE will build a user profile that tracks the user's progress and diagnoses their level. The learning content has information about the exercises, such as the goal and plan. ILDE will utilize the user profile on current and past exercises to give tailored compiler messages. For the "uninitialized object" misconception at beginner levels, the ILDE feedback

subsystem will play a video that explains the memory allocation process and how uninitialized memory affects program logic. For learners at the intermediate level, the feedback subsystem will invoke an image that shows the memory simulation of the error with a link to the related lessons.

However, programming concepts are like icebergs that hide a lot of details. On one hand, researchers of the psychology of programming urge the programming language designer to employ cognitive principles to lower the barriers for programming[32]. On the other hand, it is very important that the learner understands the ‘inner world’ of the programming language being used [27,32]. Hoc and Nguyen-Xuan added that programming acquisition is about learning basic operating rules of the processing device that underlies the language, the constraints of these operations upon the program structure, and the relation between task structure and programming language semantics [33]. So, understanding the smaller instructions that construct the programming statements and connecting these instructions to the concepts of memory and CPU operations may solve the issue of the fundamental misconceptions.

Error handling and debugging

Novices often fail to write the program structure correctly, misspell keywords, or omit or disorder the components of a program structure [14, 34]. Even modern compilers and IDEs frequently report inadequate, unclear error messages. Enhancing compiler error messages has a significant positive effect on novice programmers’ learning by reducing the number of errors, and the number of repeated errors [35-37].

A logic error is when the produced program does not perform the intended functionality, for example due to improper conversion between data types, or incorrect division of a float number by an integer [14, 34]. Ettles, Luxton-Reilly, and Denny report that misconceptions are the source of logic errors. Finding and fixing logical errors is more difficult than fixing syntax errors. Debugging skills are difficult and even good programmers often lack these skills [30].

Many tools have been integrated into IDEs to enhance messages for both syntax and logic errors. Specialized IDEs have been created for educational purposes. These efforts include PROUST[38], Merr [15], Espresso[14], BlueJ [39], TigerJython [15], Alice [40], Scratch [40], Greenfoot [40], WebTigerJython [41]. These projects are focused efforts targeting specialized aspects of programming instruction. Thus, novices still need assistance mastering basic concepts.


```

jffery@cs-1ppg00-cj ~$ cat mmr4.cpp
#include <iostream>
#include <random>
using namespace std;
int main() {
    default_random_engine *generator;
    uniform_int_distribution<int> distribution(1,20);
    cout << distribution(*generator);
}
-bash-4.2$ g++ -std=c++11 mmr4.cpp
-bash-4.2$ ./a.out
Segmentation fault
-bash-4.2$ g++ -Wall -std=c++11 mmr4.cpp
mmr4.cpp: In function 'int main()':
mmr4.cpp:7:34: warning: 'generator' may be used uninitialized in this function [-Wmaybe-uninitialized]
    cout << distribution(*generator);
                          ^
-bash-4.2$ cat mmr4.java
import java.util.Random;
class mmr4 {
    public static void main(String[]args) {
        Random numgen;
        System.out.println(numgen.nextInt()%20+1);
    }
}
-bash-4.2$ javac mmr4.java
mmr4.java:5: error: variable numgen might not have been initialized
        System.out.println(numgen.nextInt()%20+1);
                          ^
1 error
-bash-4.2$

```

Figure 3.1 Example of misconception: “uninitialized memory allocation.”

ILDE will offer informative feedback for use in introductory programming courses where the focus is on basic programming concepts. Other specialized educational IDEs prioritize support for the object-first approach that focuses on recognizing the object-oriented concepts: class, object, and members. In contrast, ILDE focuses on tailored feedback and learning of basic concepts: variable, data type, mathematical and logical operation, expression, iteration structure, if statement, and function. Other environments use block-based and graphic objects and are intended for younger learners. Those environments are designed to cover a small subset of programming concepts. Also, using graphic objects representing program constructs that are inserted and moved within a graphical environment is tricky, because of the high level of abstractions. Such visual programming environments are easier to use than text-based environments but may make it harder to learn the underlying programming concepts[32, 33, 42].

3.3 ILDE

ILDE draws upon Cognitive Load Theory, Kolb’s Experiential Learning Theory, Constructivism principles, and the Cognitive Theory of Multimedia Learning [21-24]. The learning material is

divided into phases that depend on learner performance levels. The programming concepts are highly interactive elements. The sequence of the phases depends on the interactive relation between programming elements. Phase X is not accessible to the learner until she/he reaches the intermediate or better level of performance in the phases that phase X depends on. The levels of the learner are beginner, elementary, intermediate, advanced, and expert. The learning content introduces one programming concept at a time. The first lessons in each phase are easy, with direct instructions. The lessons present worked examples, followed by guided practices and simple and detailed feedback messages to lead the learner to develop the skills and achieve fluency. Then ILDE presents the complex and related concepts together with higher-order reasoning and thinking problems. ILDE's feedback subsystem provides support and scaffolding. The sequence of instructions and lessons in each phase follows Kolb's learning cycle.

The design of multimedia learning content follows the assumptions of the Cognitive Theory of Multimedia Learning, such as the dual channels assumption, where ILDE employs verbal and visual learning content, and the active processing assumption, which entails that ILDE's learning content is a collection of self-paced training materials. Also, the same content is delivered in different ways. Formative feedback in the interactive learning environment has major potential to enhance the learning process and it can be effective in a well-defined domain [43].

Multimedia Learning Content with Visualization Tool.

The Multimedia Learning Content with Visualization (MLC) presents programming concepts using video, audio, and text to explain programming structure, meaning, usage, and behavior. MLC is integrated with a visualization tool that displays what is happening inside the computer at compilation and runtime. Moreover, ILDE has a mascot named Reynold, a squirrel character helper who presents the content and guides learners through the environment.

The following scenario illustrates how the ILDE works. The learner opens the learning content menu. The first lesson is a video showing the course project. The objectives of the first lesson are: to introduce the potential of programming to the learners and to let students recognize what they will be able to do at the end of the course. The project is keeping financial records of a poultry production farm, an inventory management problem. The lesson starts with a farmer asking the students to help him to keep the financial records of his business. Then the video presents the software requirements of the project. That is "the software must keep records of receipts, expenses, and purchases of egg production and aggregate them by day, week, month, and year". Then the lesson will show what a solution for the project looks like and how the farmer is using it.

The second lesson starts by playing a video showing students how to develop their first program. It is like the well-known “Hello world” program, but they will write the name of the poultry farm. The video shows them how to use ILDE to write the program, to compile it, to correct errors, and to run the program. In the second activity of the lesson, students will write the program by themselves. Reynold will guide them, by pointing out where to write, what button to press to compile, and so on. The task text will be available as an image lockable to the edge of the screen while they write their program. This lesson is the first stage of the learning cycle of Kolb’s experiential learning theory. It puts the learners in a concrete experiment. Then it is followed by a reflective activity; the second stage of the learning cycle. Students fill in the form of questions asking them about what they see, and how they do the task. The third activity is an explanation of how computers are dealing with their program. It is a video that shows the students a visualization of the fetch-execute cycle. It is a simple explanation of how the memory and CPU work in their first program. It shows the phases that the program passes through, from source code to an executable program.

The lessons will gradually introduce programming concepts. Following cognitive load theory recommendations, each lesson focuses on one concept at a time. In addition, it takes into consideration the different skills that must be developed, followed by exercises with no context to master these skills.

A lesson later explains the C language `for` control structure, introducing a major concept of programming. The lesson starts with a video explaining the iteration structure in the C language. It shows the anatomy of the `for` loop. For example, the anatomy of the statement

```
for(i=1; i <=7; i++) weekly_product += daily_product ;
```

can be illustrated by:

1. Assign: `i=1`;
2. Compare: `i, 7`;
3. Branch (go to) if less than: step 7.
4. ADD: add the daily product to weekly product;
5. Increment: add 1 to the counter `i`;
6. Branch (go to) to the checkpoint of the loop step 2;
7. Rest of the code.

The lesson explains the usage of the iteration control structure and connects it with a simple problem from the course project. The learners will compute the production for a week. To repeat the entries for each day of the week, they will need to use the iteration structure. MLC introduces each programming concept with a real-world context. This helps the learners to scaffold their knowledge and retain it easier.

The lesson explains language structure behavior using visualization tools. In the next activity, the lesson opens two screens. One shows a sample program's source code and the other shows the visualization of the program. An audio explanation accompanies the visualization screen, which shows how the computer interprets a language structure via a series of smaller instructions, including the variables used in a memory simulation picture and the flow of control. The visualization window shows the fetch-execute cycle for this example. It shows how the CPU works in the program and how the values of variables in memory change accordingly after each cycle. Figure 3.2 and Figure 3.3 show the visualization of memory and CPU consequently. After reviewing the example's source code execution, the learner can experiment with the source code and the visualization window reflects the change.

1	ASSIGN	i	1
2	COMPARE	i	7
3	BRANCH <=	TO STEP 7	
4	ADD	weekly_product	daily_product
5	INCREMENT	i	1
6	BRANCH	TO STEP 2	
7			

	i	3	4
weekly_product		130	
daily_product		79	




Figure 3.2 Example of memory visualization of `for(i=1; i<=7; i++) weekly_product +=daily product;`

The feedback subsystem is integrated with a customized compiler that offers tailored error messages. The MLC has exercises guided by an e-booklet. As the learner works on solving lesson problems, they may consult this e-booklet with step-by-step instructions delivered by Reynold, who highlights the relevant part of the code. Once the learner writes the source code in the editor and compiles it, the ILDE will tailor informative feedback related to the error, problem, and level of the lesson. The ILDE will also display frequent and common errors in an interactive graphical representation.



Figure 3.3 Example of CPU visualization.

The e-booklet learns the student's problem-solving skills. As it graduates from easier to more complex problems, it follows a problem-solving methodology: it teaches students how to extract the requirement and what the results should be; how to design a solution; how to implement it; how to test it. The e-booklet is an interactive screen that allows the user to write the requirements textboxes, and the design of the problem in textboxes and then write the code in the editor. The students can compare their writing of requirements and design with the correct answers. While in the editor, the customized compiler produces the error messages. Reynold talks and explains the problem and points to relevant parts of the screen.

The customized compiler takes the learner level and the problem the learner works on and gives tailored informative messages about the syntax errors. The feedback subsystem opens the related lesson video for common reported errors.

At the first level, students need intensive feedback while they work on tiny programs. Knowing the problem and the expected results enables the feedback subsystem to give precise tailored messages, and also connect these messages with the related lessons. The customized compiler compares the part of the student's code which is located before the detected error with possible correct solutions. This allows it to predict what statement the student is trying to write and explain what the error is in it.

The help menu has pictures of the programming statements syntax and function signatures. It enables the learner to dock these pictures on the edges of the working area of ILDE. Clicking on a syntax picture copies the content as text that can be pasted, adding a statement into the editor. Moreover, the feedback subsystem uses comic pictures accompanied by sounds when pointing to the errors. For

example, common errors novices frequently fall into are missing semicolons, or missing braces. When this occurs, an animated icon with music sounds as the code is corrected. This will make an unstressed and memorable learning experience.

3.5 Conclusion

ILDE is a specialized environment for learning and teaching novice programmers. The innovative aspects of ILDE are:

- The integration of learning content into the environment using modern pedagogical and cognitive psychology practice.
- Visualization of memory allocation and CPU operations. That makes it possible to explain the hidden parts of programming and visualize dynamically what happens inside the program.
- Feedback subsystem with customized compiler. Integration of learning content with ILDE enables the feedback subsystem to know the context of problems that students work on. This enables ILDE to give tailored error messages to the learner.

ILDE is designed for C/C++ which is difficult to learn and has a complex syntax. Their relative difficulty makes the need for an educational learning development environment greater, rather than lesser than is needed in “easier” first languages.

Finally, the visualization features of ILDE may be used to learn other topics, such as data structures, algorithms, and programming language design. Also, with selection of an appropriate subset of the curriculum, ILDE may be used effectively in K-12 computer education.

Chapter 4: Analysis of Syntax Error Messages from the Learner's Perspective

To explore the problem of unhelpful syntax error messages, let us examine the behavior of two popular compilers: GNU GCC version 10.3.1 (GCC) and Microsoft Visual C++ version 2019 (VC++), on a set of erroneous programs for common errors in the structures of the function body, the if statement, and the for statement that we present in Figure 4.1, Figure 4.7, and Figure 4.12.

4.1 Analysis of common errors in the syntax of the function body

One of the common errors of the syntax of the function body is the unbalanced curly bracket. Let us study how the compilers report this error on different programs, such as in Figure 4.1.

First, consider prog1.cpp. The program has a syntax error: unbalanced curly brackets. The close curly bracket '}' for the open curly bracket '{' on line 5 is missing. GCC reports the error in a good error message, as shown in Figure 4.2, it says: "error: expected '}' at the end of input. Note: to match this '{' in line 5". VC++ says for the same error, "'{' no matching token found." Both reports are correct, and the messages are understandable, and they may help the novice find the error and fix the problem. Unfortunately, both compilers report misleading messages for the same type of error in other programs.

Second, GCC and VC++ report the error of prog2.cpp as shown in Figure 4.3 with poor quality error messages. The GCC message says: "expected primary-expression before '}' token" in line 7. It then says "at global level, cout does not name a type" in line 8, and then it says "expected declaration before '}' token" in line 9 and in line 10. VC++ lists 11 errors for the same program. Some of them are: "expected a statement" in line 7, "this declaration has no storage class or type specifier" in line 8, and "expected a ';' in line 8. The content of the messages is incorrect and is not understandable. They use compiler writers' jargon such as "primary-expression," and "storage class or type specifier."

Third, analyzing the quality of syntax error messages for prog3.cpp and prog4.cpp as shown in Figure 4.4 and Figure 4.5, we will find that both GCC and VC++ reports are poor. The messages are incorrect and aren't understandable, and it may not help learners find or fix errors.

<pre> Prog1.cpp: 1: #include<iostream> 2: using namespace std; 3: int main() 4: {} 5: int recursive(int x){ 6: if (x > 1){ 7: return x * recursive(x -1); 8: } else { 9: return 1; 10: }</pre>	<pre> Prog2.cpp: 1: #include<iostream> 2: using namespace std; 3: int main() 4: { 5: int x=1; 6: for(x=1; x<5;x++) 7: } 8: cout<<"x"<<endl; 9: } 10: }</pre>
<pre> prog3.cpp: 1: #include<iostream> 2: using namespace std; 3: int main() 4: { 5: int x=1; 6: int c=x+10;} 7: for(int x=1;x<5) 8: { 9: cout<<" x"<<endl; 10: x++; 11: } 12: }</pre>	<pre> Prog4.cpp: 1: #include<iostream> 2: using namespace std; 3: int f(int); 4: int main() 5: { 6: int x,y, res; 7: cin>>x>>y; 8: ++x; 9: y=x+9; 10: res=f(x)+f(y); 11: ++res; 12: cout<<"result="<<res; 13:) 14: int f(int x){ return x*x;}</pre>

Figure 4.1 Some C++ programs with common syntax errors of novice programmers: unbalanced curly brackets.

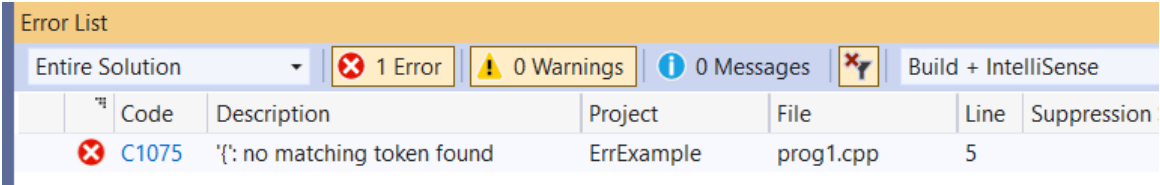
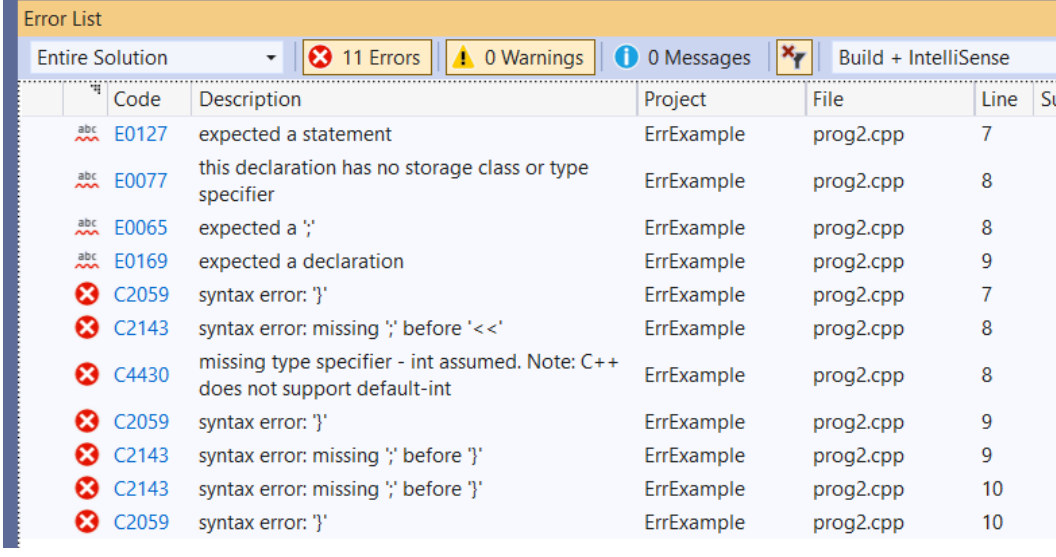
<p>GNU GCC report:</p> <pre> [sanaadmin@shams ErrExamples3]\$ g++ prog1.cpp prog1.cpp: In function 'int recursive(int)': prog1.cpp:10:1: error: expected '}' at end of input 10 } ^ prog1.cpp:5:22: note: to match this '{' 5 int recursive(int x){ ^ [sanaadmin@shams ErrExamples3]\$</pre>
<p>Microsoft Visual C++ report:</p>  <p>The screenshot shows the 'Error List' window in Visual Studio. It displays one error: 'C1075 '{': no matching token found' in the file 'prog1.cpp' at line 5. The error is highlighted in red. The window also shows '1 Error', '0 Warnings', and '0 Messages'.</p>

Figure 4.2 Good quality syntax error messages reported by GCC and VC++ on compilation of prog1.cpp in Figure 4.1

GNU GCC report:

```
[sanaadmin@shams ErrExamples3]$ g++ prog2.cpp
prog2.cpp: In function 'int main()':
prog2.cpp:7:3: error: expected primary-expression before '}' token
  7 |     }
    |     ^
prog2.cpp: At global scope:
prog2.cpp:8:4: error: 'cout' does not name a type
  8 |     cout<<"x"<<endl;
    |     ^~~~~
prog2.cpp:9:3: error: expected declaration before '}' token
  9 |     }
    |     ^
prog2.cpp:10:1: error: expected declaration before '}' token
 10 | }
    | ^
```

Microsoft Visual C++ report:



Code	Description	Project	File	Line	St
E0127	expected a statement	ErrExample	prog2.cpp	7	
E0077	this declaration has no storage class or type specifier	ErrExample	prog2.cpp	8	
E0065	expected a ;'	ErrExample	prog2.cpp	8	
E0169	expected a declaration	ErrExample	prog2.cpp	9	
C2059	syntax error: }'	ErrExample	prog2.cpp	7	
C2143	syntax error: missing ;' before '<<'	ErrExample	prog2.cpp	8	
C4430	missing type specifier - int assumed. Note: C++ does not support default-int	ErrExample	prog2.cpp	8	
C2059	syntax error: }'	ErrExample	prog2.cpp	9	
C2143	syntax error: missing ;' before }'	ErrExample	prog2.cpp	9	
C2143	syntax error: missing ;' before }'	ErrExample	prog2.cpp	10	
C2059	syntax error: }'	ErrExample	prog2.cpp	10	

Figure 4.3 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog2.cpp in Figure 4.1

GNU GCC report:

```
[sanaadmin@shams ErrExamples3]$ g++ prog3.cpp
prog3.cpp:7:3: error: expected unqualified-id before 'for'
  7 |     for(int x=1;x<5;)
    |     ^~~~~
prog3.cpp:7:15: error: 'x' does not name a type
  7 |     for(int x=1;x<5;)
    |               ^
prog3.cpp:7:19: error: expected unqualified-id before ')' token
  7 |     for(int x=1;x<5;)
    |                   ^
prog3.cpp:12:1: error: expected declaration before '}' token
 12 | }
    | ^
```

Microsoft Visual C++ report:

Error List						
Entire Solution		9 Errors	0 Warnings	0 Messages	Build + IntelliSens	
	Code	Description	Project	File	Line	
	E0169	expected a declaration	ErrExample	prog3.cpp	8	
	C2059	syntax error: 'for'	ErrExample	prog3.cpp	8	
	C2143	syntax error: missing ')' before ';'	ErrExample	prog3.cpp	8	
	C2143	syntax error: missing ';' before '<'	ErrExample	prog3.cpp	8	
	C4430	missing type specifier - int assumed. Note: C++ does not support default-int	ErrExample	prog3.cpp	8	
	C2059	syntax error: '}'	ErrExample	prog3.cpp	8	
	C2447	'{: missing function header (old-style formal list?)	ErrExample	prog3.cpp	9	
	C2059	syntax error: '}'	ErrExample	prog3.cpp	13	
	C2143	syntax error: missing ';' before '}'	ErrExample	prog3.cpp	13	

Figure 4.4 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog3.cpp in Figure 4.1

GNU GCC report:

```
[sanaadmin@shams ErrExamples3]$ g++ prog4.cpp
prog4.cpp: In function 'int main()':
prog4.cpp:10:15: error: expected ')', before '}' token
   10 |     res=f(x)+f(y);
       |               ^
       |               )
prog4.cpp: At global scope:
prog4.cpp:11:3: error: expected unqualified-id before '++' token
   11 |     ++res;
       |     ^~
prog4.cpp:12:3: error: 'cout' does not name a type
   12 |     cout<<"result="<<res;
       |     ^~~~
prog4.cpp:13:1: error: expected unqualified-id before ')' token
   13 | )
       | ^
[sanaadmin@shams ErrExamples3]$
```

Microsoft Visual C++ report:

Error List						
Entire Solution		11 Errors	0 Warnings	0 Messages	Build + IntelliSense	
	Code	Description	Project	File	Line	Suppress
	E0018	expected a '}'	ErrExample	prog4.cpp	10	
	E0169	expected a declaration	ErrExample	prog4.cpp	11	
	E0077	this declaration has no storage class or type specifier	ErrExample	prog4.cpp	12	
	E0065	expected a ';'	ErrExample	prog4.cpp	12	
	E0169	expected a declaration	ErrExample	prog4.cpp	13	
	C2143	syntax error: missing ')' before '}'	ErrExample	prog4.cpp	10	
	C2143	syntax error: missing ';' before '}'	ErrExample	prog4.cpp	10	
	C2059	syntax error: '++'	ErrExample	prog4.cpp	11	
	C2143	syntax error: missing ';' before '<<'	ErrExample	prog4.cpp	12	
	C4430	missing type specifier - int assumed. Note: C++ does not support default-int	ErrExample	prog4.cpp	12	
	C2059	syntax error: '}'	ErrExample	prog4.cpp	13	

Figure 4.5 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog4.cpp in Figure 4.1

Analyzing the previous examples from a parsing techniques perspective, it is apparent that the quality of error messages is bad when a parser recognizes part of the source code as a function body. So, the rest of the source code is at the global scope of the program.

Figure 4.7 shows drawings that clarify the boundaries of function bodies for the previous examples. Considering prog2.cpp from Figure 4.7, we will find GCC reported that in line 8 of the source code: “At global scope: error: ‘cout’ does not name a type”, and in line 9 “error: expected declaration before ‘}’ token.” Also, VC++ reported that for line 8 with a red line under the ‘cout’ : “ this declaration has no storage class or type specifier” and another error for the same line “expected a ‘;’.” However, line 8 itself has no syntax error if it is placed inside the boundaries of a function body, while for prog1.cpp both GCC and VC++ reported good error messages indicating that there are unbalanced curly brackets because the parser does not reduce the function body yet when it counters the error.

GCC’s message is a good message for prog1.cpp, although it contains a compiler writer jargon “primary-expression,” and this sometimes could be a distraction even if the message is correct. At this point, it is time to focus on how a parser recognizes the source code and how they report the errors accordingly. If the compiler can tell users how it has recognized the function boundaries and how these statements are outside the function boundaries, it may leverage the quality of syntax error messages for this type of syntax error.

<pre> Prog1.cpp: 1: #include<iostream> 2: using namespace std; 3: int main() 4: {} 5: int recursive(int x){ 6: if (x > 1){ 7: return x * recursive(x -1); 8: } else { 9: return 1; 10: } </pre>	<pre> Prog2.cpp: 1: #include<iostream> 2: using namespace std; 3: int main() 4: { 5: int x=1; 6: for(x=1; x<5;x++) 7: } 8: cout<<"x"<<endl; 9: } 10: } </pre>
<pre> prog3.cpp: 1: #include<iostream> 2: using namespace std; 3: int main() 4: { 5: int x=1; 6: int c=x+10;} 7: for(int x=1;x<5;) 8: { 9: cout<<" x="<<endl; 10: x++; 11: } 12: } </pre>	<pre> Prog4.cpp: 1: #include<iostream> 2: using namespace std; 3: int f(int); 4: int main() 5: { 6: int x,y, res; 7: cin>>x>>y; 8: ++x; 9: y=x+9; 10: res=f(x)+f(y); 11: ++res; 12: cout<<"result="<<res; 13: } 14: int f(int x){ return x*x;} </pre>

Figure 4.6 The C++ programs From Figure 4.1 with drawings that clarify how a parser recognizes the function body boundaries.

4.2 Analysis of common errors in the syntax of the if statement

GCC and VC++ reports for syntax errors in the control structures show that they generate poor error messages for simple and common syntax errors. For example, Figures 4.8-4.11 show reports from both compilers for erroneous programs where an if statement's header is missing its parentheses. The programs are listed in Figure 4.7. One may wonder how confusing and misleading these messages are, despite most of them only having this single error. It is worse when there is one or more errors in the expression inside the header parentheses besides missing one or both of its parentheses.

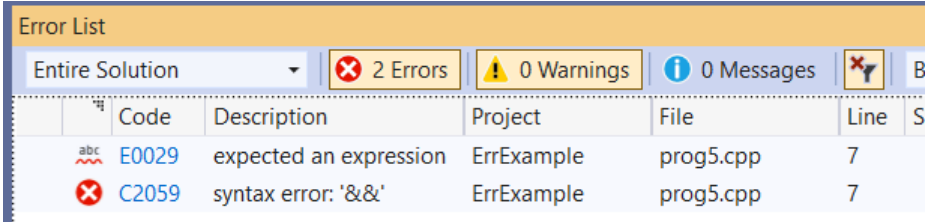
<pre> prog5.cpp 1: #include<iostream> 2: using namespace std; 3: int main() 4: { 5: int a,b; 6: cin>>a>>b; 7: if(a*b<=200) && (b<200) 8: cout<<"You win!"<<endl; 9: }</pre>	<pre> prog6.cpp 1: #include<iostream> 2: using namespace std; 3: int main() 4: { 5: int a,b; 6: cin>>a>>b; 7: if(a*b<=200) & (b<200) 8: cout<<"You win!"<<endl; 9: }</pre>
<pre> prog7.cpp: 1: #include<iostream> 2: using namespace std; 3: int main() 4: { 5: int a,b; 6: cin>>a>>b; 7: if(a*b<=200) (b<200) 8: cout<<"You win!"<<endl; 9: }</pre>	<pre> prog8.cpp 1: #include<iostream> 2: using namespace std; 3: int main() 4: { 5: int var1, var2, var3; 6: int sum = var1 + var2 + var3; 7: int avg = sum / 3; 8: cout << "The sum is " << sum << 9: endl; 10: cout << "The average is " << avg << 11: endl; 12: if (var1 < (var2 && var3) cout << "The smallest number is " << var1; }</pre>

Figure 4.7 C++ programs with common syntax errors of novice programmers if statement header.

GNU GCC report:

```
[sanaadmin@shams ErrExTest]$ g++ prog5.cpp
prog5.cpp: In function 'int main()':
prog5.cpp:7:17: error: expected identifier before '(' token
   7 |     if(a*b<=200)&&(b<200)
     |                   ^
[sanaadmin@shams ErrExTest]$
```

Microsoft Visual C++ report:



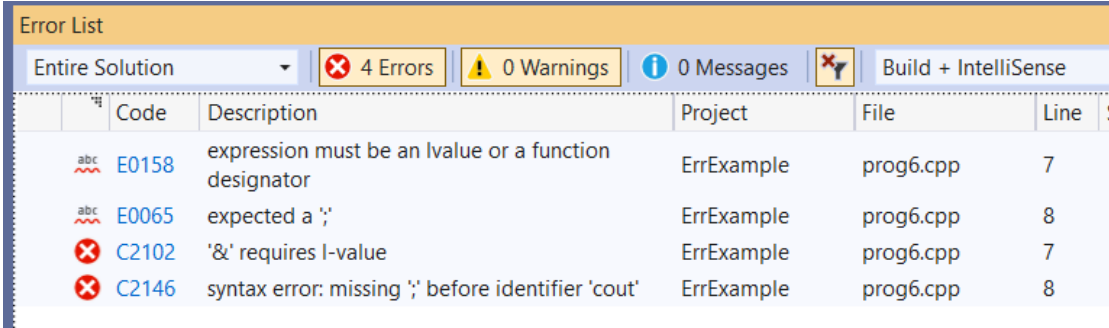
Code	Description	Project	File	Line
E0029	expected an expression	ErrExample	prog5.cpp	7
C2059	syntax error: '&&'	ErrExample	prog5.cpp	7

Figure 4.8 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog5.cpp in Figure 4.7

GNU GCC report:

```
[sanaadmin@shams ErrExTest]$ g++ prog6.cpp
prog6.cpp: In function 'int main()':
prog6.cpp:7:18: error: lvalue required as unary '&' operand
   7 |     if(a*b<=200)&(b<200)
     |                   ~^~~~~
[sanaadmin@shams ErrExTest]$
```

Microsoft Visual C++ report:



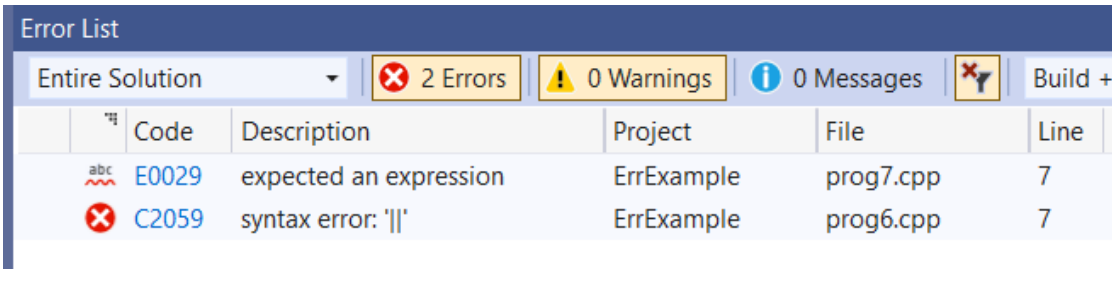
Code	Description	Project	File	Line
E0158	expression must be an lvalue or a function designator	ErrExample	prog6.cpp	7
E0065	expected a ;'	ErrExample	prog6.cpp	8
C2102	'&' requires l-value	ErrExample	prog6.cpp	7
C2146	syntax error: missing ; before identifier 'cout'	ErrExample	prog6.cpp	8

Figure 4.9 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog6.cpp in Figure 4.7

GNU GCC report:

```
[sanaadmin@shams ErrExTest]$ g++ prog7.cpp
prog7.cpp: In function 'int main()':
prog7.cpp:7:15: error: expected primary-expression before '||' token
   7 |     if(a*b<=200)|| (b<200)
     |                   ^~
[sanaadmin@shams ErrExTest]$
```

Microsoft Visual C++ report:



Code	Description	Project	File	Line
E0029	expected an expression	ErrExample	prog7.cpp	7
C2059	syntax error: '\ \ '	ErrExample	prog6.cpp	7

Figure 4.10 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog7.cpp in Figure 4.7

GNU GCC report:

```
[sanaadmin@shams ErrExTest]$ g++ prog8.cpp
prog8.cpp: In function 'int main()':
prog8.cpp:10:6: warning: init-statement in selection statements only
7' or '-std=gnu++17'
 10 |   if (var1 < (var2 && var3)
    |         ^~~~
prog8.cpp:10:27: error: expected ';' before 'cout'
 10 |   if (var1 < (var2 && var3)
    |                       ^
    |                       |
    |                       |
 11 |   cout << "The smallest number is " << var1;
    |   ~~~~
prog8.cpp:12:1: error: expected primary-expression before '}' token
 12 | }
    | ^
prog8.cpp:11:43: error: expected ')' before '}' token
 11 | cout << "The smallest number is " << var1;
    |                                           ^
    |                                           )
 12 | }
    | ~
prog8.cpp:10:5: note: to match this '('
 10 |   if (var1 < (var2 && var3)
    |     ^
prog8.cpp:12:1: error: expected primary-expression before '}' token
 12 | }
    | ^
[sanaadmin@shams ErrExTest]$
```

Microsoft Visual C++ report:

Error List						
Entire Solution		5 Errors	3 Warnings	0 of 3 Messages	Build + IntelliSense	
	Code	Description	Project	File	Line	
	E0018	expected a ')'	ErrExample	prog8.cpp	11	
	E0127	expected a statement	ErrExample	prog8.cpp	12	
	Int-logical...	Using logical '&&' when bitwise '&' was probably intended.	ErrExample	prog8.cpp	10	
	C4804	'<: unsafe use of type 'bool' in operation	ErrExample	prog8.cpp	10	
	C2146	syntax error: missing ';' before identifier 'cout'	ErrExample	prog8.cpp	11	
	C4552	'<: result of expression not used	ErrExample	prog8.cpp	10	
	C2429	language feature 'init-statements in if/switch' requires compiler flag '/std:c++17'	ErrExample	prog8.cpp	11	
	C2059	syntax error: '<<'	ErrExample	prog8.cpp	11	

Figure 4.11 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog8.cpp in Figure 4.7

4.3 Analysis of common errors in the syntax of the for statement

Loop control structures, especially the `for` statement, are problematic for learners. Many misconceptions are related to them, but the compilers often do not help. On the contrary, they generate bad error messages for the related common syntax errors. Figures 4.13 – 4.16 show compiler behaviors for the programs in Figure 4.12. The programs have missing parentheses or writing ‘,’ instead of ‘;’ inside the `for` statement’s header. When the expression inside the header is wrongly connected to the statements of the body, the compilers behave in ways that are confusing to learners. Notice how both generate lists of incorrect error messages in Figure 4.16 for the `prog12.cpp`.

<pre> prog9.cpp 1: #include<iostream> 2: using namespace std; 3: int main() 4: { 5: int x=10;int y=8; 6: for(x+0=2 y<10) 7: { 8: cout<<"hello"; 9: } 10: }</pre>	<pre> prog10.cpp 1: #include<iostream> 2: using namespace std; 3: int main() 4: { 5: int x,y,sum=0; 6: for(x=0, x<9,x++) 7: cin>>y; 8: sum=sum+y; 9: cout<<"sum="<<sum; 10: }</pre>
<pre> prog11.cpp: 1: #include<iostream> 2: using namespace std; 3: int main() 4: { 5: int x,y,sum=0; 6: for(x=0, x<9,x+++)</pre> <pre> 7: cin>>y; 8: sum=sum+y; 9: cout<<"sum="<<sum; 10: }</pre>	<pre> prog12.cpp 1: #include<iostream> 2: using namespace std; 3: int main() 4: { 5: int x,y; 6: for(x=1,y<10,x+ 7: cin>>y; 8: }</pre>

Figure 4.12 C++ programs with common syntax errors of novice programmers, for statement header.

GNU GCC report:

```
[sanaadmin@shams ErrExTest]$ g++ prog9.cpp
prog9.cpp: In function 'int main()':
prog9.cpp:6:7: error: lvalue required as left operand of assignment
   6 |     for(x+0=2 || y<10)
     |         ~^~
prog9.cpp:10:1: error: expected primary-expression before '}' token
  10 | }
     | ^
prog9.cpp:9:3: error: expected ';' before '}' token
   9 | }
     |   ^
  10 | }
     |   ~
prog9.cpp:10:1: error: expected primary-expression before '}' token
  10 | }
     | ^
prog9.cpp:9:3: error: expected ')' before '}' token
   9 | }
     |   ^
  10 | }
     |   ~
prog9.cpp:6:5: note: to match this '('
   6 |     for(x+0=2 || y<10)
     |         ^
prog9.cpp:10:1: error: expected primary-expression before '}' token
  10 | }
     | ^
[sanaadmin@shams ErrExTest]$
```

Microsoft Visual C++ report:

Error List									
Entire Solution		6 Errors		0 Warnings		0 Messages		Build + IntelliSens	
	Code	Description	Project	File	Line				
abc	E0137	expression must be a modifiable lvalue	ErrExample	prog9.cpp	6				
abc	E0065	expected a ';' before '}'	ErrExample	prog9.cpp	6				
	C2106	'=': left operand must be l-value	ErrExample	prog9.cpp	6				
	C2143	syntax error: missing ';' before '}'	ErrExample	prog9.cpp	6				
	C2143	syntax error: missing ';' before '{'	ErrExample	prog9.cpp	7				
	C2143	syntax error: missing ')' before '{'	ErrExample	prog9.cpp	7				

Figure 4.13 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog9.cpp in Figure 4.12

GNU GCC report:

```
[sanaadmin@shams ErrExTest]$ g++ prog10.cpp
prog10.cpp: In function 'int main()':
prog10.cpp:6:19: error: expected ';' before ')' token
   6 |     for(x=0, x<9,x++)
     |                   ^
prog10.cpp:9:20: error: expected ')' before ';' token
   9 |     cout<<"sum="<<sum;
     |                   ^
prog10.cpp:6:6: note: to match this '('
   6 |     for(x=0, x<9,x++)
     |     ^
[sanaadmin@shams ErrExTest]$
```

Microsoft Visual C++ report:

Code	Description	Project	File	Line	S
E0065	expected a ';'	ErrExample	prog10.cpp	6	
C2143	syntax error: missing ';' before ')'	ErrExample	prog10.cpp	6	
C2146	syntax error: missing ')' before identifier 'cout'	ErrExample	prog10.cpp	9	
C2059	syntax error: ';'	ErrExample	prog10.cpp	9	

Figure 4.14 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog9.cpp in Figure 4.12

GNU GCC report:

```
[sanaadmin@shams ErrExTest]$ g++ prog11.cpp
prog11.cpp: In function 'int main()':
prog11.cpp:6:20: error: expected primary-expression before ')' token
   6 |     for(x=0, x<9,x++)
     |                   ^
prog11.cpp:9:20: error: expected ')' before ';' token
   9 |     cout<<"sum="<<sum;
     |                   ^
prog11.cpp:6:6: note: to match this '('
   6 |     for(x=0, x<9,x++)
     |     ^
[sanaadmin@shams ErrExTest]$
```

Microsoft Visual C++ report:

Code	Description	Project	File	Line	S
E0029	expected an expression	ErrExample	prog11.cpp	6	
C2059	syntax error: ')'	ErrExample	prog11.cpp	6	

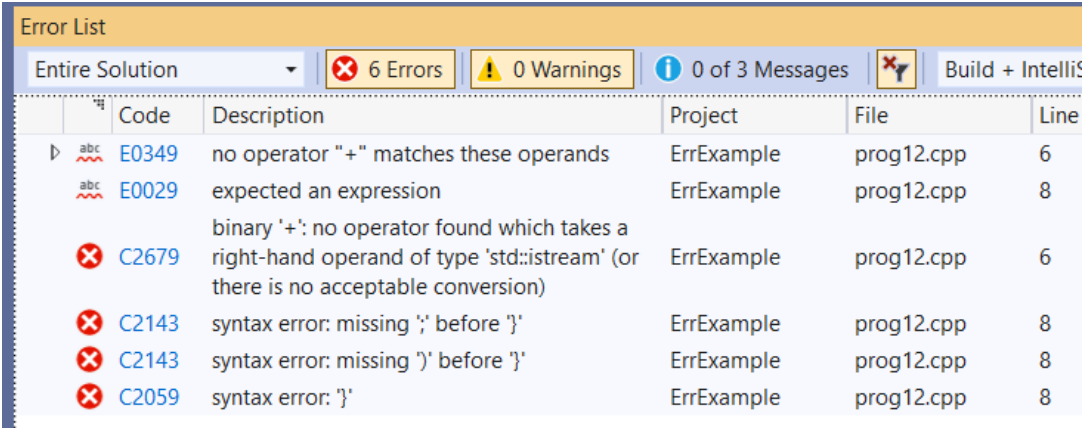
Figure 4.15 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog11.cpp in Figure 4.12

GNU GCC report:

```
[sanaadmin@shams ErrExTest]$ g++ prog12.cpp
prog12.cpp: In function 'int main()':
prog12.cpp:6:17: error: no match for 'operator+' (operand types are 'int' and 'std::istream' {aka 'std::basic_istream<char>'})
   6 |         for(x=1,y<10,x+
     |                       ~^
     |                       |
     |                       int
   7 |     cin>>y;
     |     ~~~
     |     |
     |     std::istream {aka std::basic_istream<char>}
prog12.cpp:6:17: note: candidate: 'operator+(int, int)' (built-in)
   6 |         for(x=1,y<10,x+
     |                       ~^
   7 |     cin>>y;
     |     ~~~

prog12.cpp:6:17: note: no known conversion for argument 2 from 'std::istream' {aka 'std::basic_istream<char>' } to 'int'
In file included from /usr/include/c++/10/bits/stl_algobase.h:67,
                 from /usr/include/c++/10/bits/char_traits.h:39,
                 from /usr/include/c++/10/ios:40,
                 from /usr/include/c++/10/ostream:38,
                 from /usr/include/c++/10/iostream:39,
                 from prog12.cpp:1:
/usr/include/c++/10/bits/stl_iterator.h:533:5: note: candidate: 'template<class _Iterator> std::reverse_iterator<_Iterator> std::operator+(const std::reverse_iterator<_Iterator>::difference_type, const std::reverse_iterator<_Iterator>&)'
  533 |     operator+(typename reverse_iterator<_Iterator>::difference_type __n,
      |     ^~~~~~
/usr/include/c++/10/bits/stl_iterator.h:533:5: note: template argument deduction/substitution failed:
prog12.cpp:7:2: note: 'std::istream' {aka 'std::basic_istream<char>' } is not derived from 'const std::reverse_iterator<_Iterator>'
   7 |     cin>>y;
     |     ^~~
```

Microsoft Visual C++ report:



Code	Description	Project	File	Line
E0349	no operator "+" matches these operands	ErrExample	prog12.cpp	6
E0029	expected an expression	ErrExample	prog12.cpp	8
C2679	binary '+': no operator found which takes a right-hand operand of type 'std::istream' (or there is no acceptable conversion)	ErrExample	prog12.cpp	6
C2143	syntax error: missing ';' before '}'	ErrExample	prog12.cpp	8
C2143	syntax error: missing ')' before '}'	ErrExample	prog12.cpp	8
C2059	syntax error: '}'	ErrExample	prog12.cpp	8

Figure 4.16 Bad quality syntax error messages reported by GCC and VC++ on compilation of prog12.cpp in Figure 4.12

4.4 Conclusion

This chapter studied the behavior of mainstream compilers: GNU GCC version 10.3.1 (GCC) and Microsoft Visual C++ version 2019 (VC++), on a set of erroneous programs for common errors in the structures of the function body, the `if` statement, and the `for` statement. It showed how these compilers report common and simple errors with misleading and unclear error messages. Next, a new solution is presented to generate better error messages.

Chapter 5: Engineering a Compiler for Better Error Messages

5.1 Introduction

Chapter Four presents the research problem. This chapter presents a solution design. Furthermore, the next chapter presents an evaluation of the design.

5.2 Compiler

The compiler is a computer program that translates the source code of a high-level language into machine code to create an executable program[44]. Compiler architecture mainly consists of the following passes, shown in Figure 5.1:

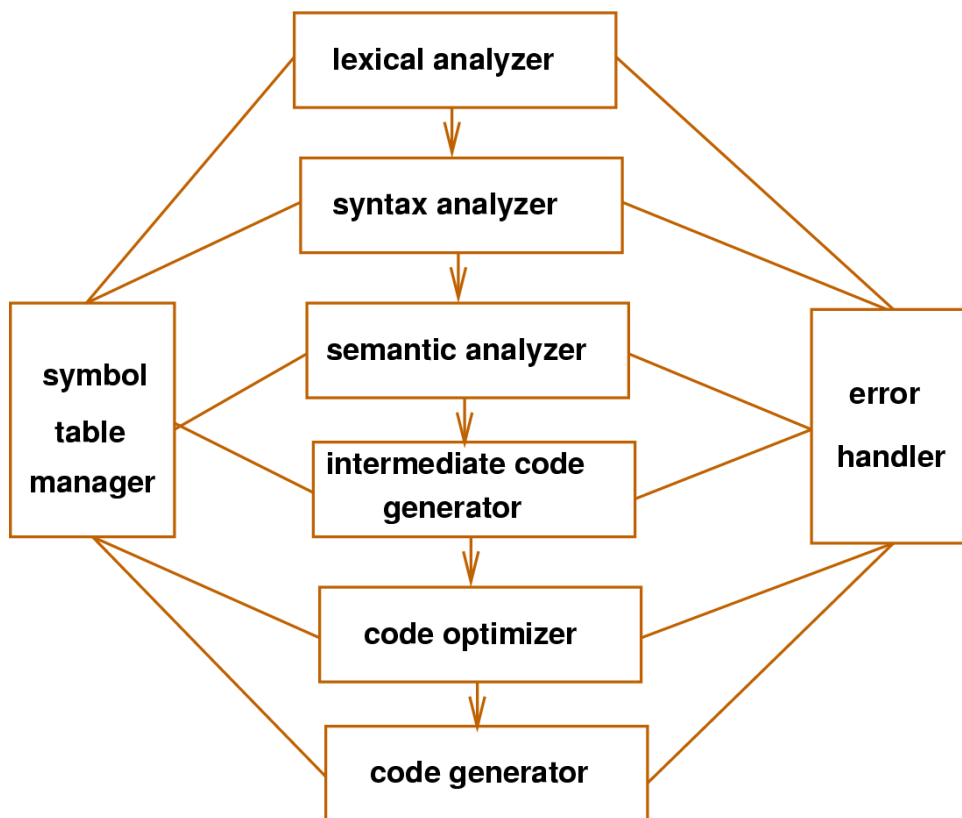


Figure 5.1 Compiler architecture

1. **The scanner** is a lexical analyzer that scans the source code characters and groups them into tokens. The scanner will report errors if characters are not allowed based on the program's syntax. These types of errors are called **lexical errors**.
2. **The Parser** is responsible for the syntax analysis. It takes the input from the lexical analyzer and builds the syntax tree depending on the language's grammar. In other words, it determines if the source code is in the appropriate order based on the production rules of the

language's grammar. If the tokens from the scanner are in the correct order, the parser generates an abstract syntax tree. Otherwise, it reports errors in the source code syntax. In advanced compilers, it recovers and continues the parsing process to report all errors in the input source code.

An abstract syntax tree (AST) represents the source code that conforms to the language's grammar. Each tree node represents a language construct. The compiler uses the AST in the following passes, so it does not reparse the source code.

The parser detects and reports **syntax errors**. Syntax errors arise due to many reasons such as incorrect order of tokens, missing punctuations, or misspelling keywords. Advanced compilers have error-handling systems that implement one or more error recovery mechanisms.

3. **The semantic analyzer** performs several tasks including type checking. The compiler creates a table of names (variables, constants, etc.) used in the programs. The semantic analyzer's job is to ensure that all the names are used in contexts where they are legal for the operations in which they are being used. It checks their types to determine what operations are being performed. After type checking, the semantic analyzer annotates the AST with extra information about where variables are declared and their type. Semantic analyzer reports the type-related errors, or what are called **semantic errors**.
4. **The intermediate code generator** conveys all information derived from previous passes about the program being compiled in a representation. This representation is called the intermediate representation. This pass calculates memory locations for all variables, control flow, and function calls. It builds a list of machine-independent intermediate code instructions.
5. **The final code generator** prepares the actual bytecode from the list of intermediate code instructions in a file format ready to load and execute.

5.3 Parser

This section briefly discusses how the parser detects syntax errors and explains why and how the proposed 3-phase parsing techniques improve syntax error messages. As discussed in the previous section, if the parser decides that the input is a valid program, it builds the abstract syntax tree that is used by the following passes of the compiler; otherwise, it announces that the program has syntax errors. Context-free grammar notation is used to express the language's correct syntax.

The most challenging part of building a parser is finding a sequence of production rules from the language grammar to apply to verify the input source code (string). Since this is one of the most critical issues in software production, many parsing techniques have been developed and tested, and a considerable amount of theory has accumulated. This research uses LALR(1) parsing techniques and a tool called Bison to generate the parsers and test the presented grammar and the related automaton. LALR is a widely used, powerful, efficient, and practical parsing technique. LALR(1), LR(0), and LR(1) are all from the LR family, which are bottom-up parsers. YACC and Bison generate LR parsers and employ LALR (1) parser tables.

LR methods are based on the combination of two ideas [45]:

- 1) They construct a finite-state automaton and read input from left to right. The aim is to find a valid production rule as efficiently as possible.
- 2) They start the automaton with the start rule of the grammar and only consider right-hand sides that could be derived from the start symbol to find a left-most reducible substring. The resulting automaton is started in its initial state and stopped at the accepting state when it recognizes the end of the production rule at the right. LALR uses the shift-reduce technique and constructs a parsing table to help parse the input string in linear time cost.

The LR Parsing Algorithm [46]:

As shown in Figure 5.2, the model of an LR parser consists of an input, an output, a stack, a driver program, and a parsing table with two parts: action and goto. The parsing program takes tokens from the scanner one at a time and stores the state number and the deriving information of the nonterminal and terminal into the stack. So, it first pushes onto the stack the initial State, then proceeds depending on the current token from the scanner. It consults the parsing table to determine what to do next: shift or reduce. The table has two parts, action, and goto. So, from Figure 5.3, if the top of the stack is state 2, and the current token is *, the action part tells the program to shift s_7 , which means pushing State seven into the top of the stack. The program should do one of the following actions while the top of the stack is state m , and the current token is a_i depending on the parsing table:

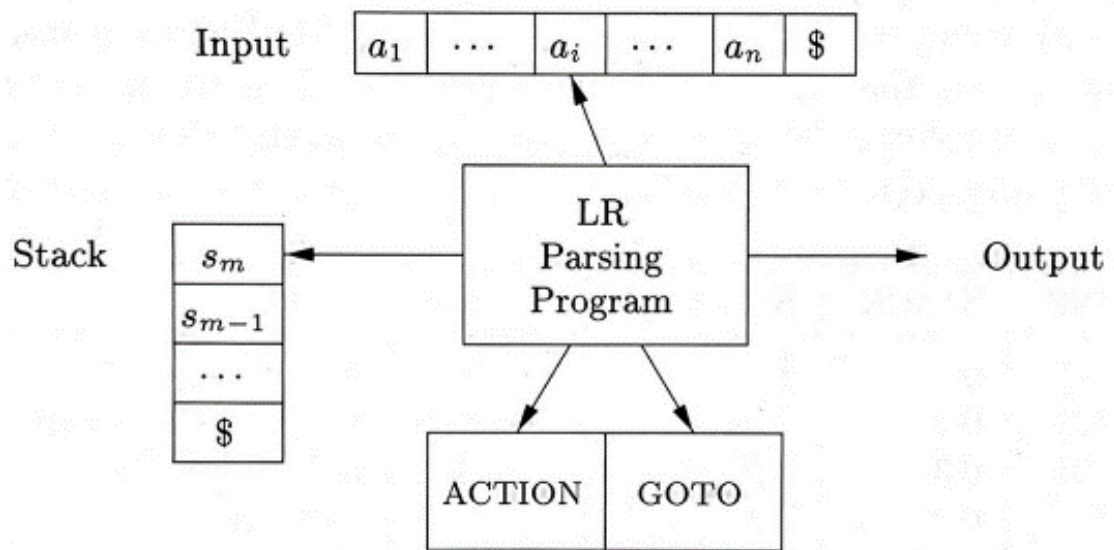


Figure 5.2 Model of an LR parser [47]

1. Shift: push the current token and the state number into the top of the stack.
2. Reduce: replace the right side of the grammar with the left side ($A \rightarrow \beta$). In other words, pop up the input, corresponding state numbers, and related derivation from the stack; $2*|\beta|$.

In both shift and reduce actions, the stack configuration will change. Alternatively, the parser tables terminate parsing with one of the following actions.

3. Accept the input source code, and parsing is completed. If the parsing program reaches the end of the input and the stack is empty. It means the input source code was derived correctly from the production rules.
4. Error, in this case, when the program reaches a state where no action is related to the current token. The cell in the action table is empty. Or the end of the input is reached and the stack is not empty. It will call the error function and report errors in the input.

Example of parsing erroneous string for the grammar in Figure 5.3 and using the LALR parser in the exact figure. Input is $5+*9$:

Step 1: The automaton starts from the initial State, s_0 , and a first token is a number, so the action is shifted and goes to state 4.

Step 2: the stack has State 0, number, State 4, and the next token is $+$, so the action is reduce using rule 5 ($F \rightarrow \text{number}$), pop from the stack State 4 and number, and push F. The stack has s_0 and F, so the GOTO is to push State 3.

Step 3: the stack has State 0, F, and State 3, so the action is reduce using rule 4 ($T \rightarrow F$), and pop from the stack F, State 3, and push T, the stack now has State 0 and T, the GOTO is to push State 2.

Step 4: the stack has State 0, T, and State 2, so the action is reduce using rule 2 ($E \rightarrow T$), pop State 2 and T, and push E. the GOTO is to push State 1.

Step 5: the stack has State 0, and E, State 1, and the action is to shift the + and enter State 5.

Step 6: the stack has State 0, E, State 2, and State 5, and the next token is *, but the table slot for the * in State 5 is empty, so an error is detected.

	Action				Goto		
	+	*	number	\$	E	T	F
0			s4		1	2	3
1	s5			acc			
2	r2	s7		r2			
3	r4	r4		r4			
4	r5	r5		r5			
5			s4			6	3
6	r1	s7		r1			
7			s4				8
8	r3	r3		r3			

E' : E
;
1. E : E + T
2. T
;
3. T : T * F
4. F
;
5. F : number
;

Figure 5.3 Grammar for an expression and the LALR parsing table for the grammar on the left. Where s_i means to shift and stack State i , r_j means to reduce by production numbered j , acc means to accept, and blank means error.

5.4 Error detection

An LR parser detects an error when the parsing program consults the parsing table and finds a blank entry. So, the LR parser reports an error because there is no valid continuation related to the current token, and as a result, for the portion of the scanned input. While all LR parsers will never shift an erroneous input into the stack, the simple LR(SLR) and LALR may make several reduction actions before reporting an error. The main point here is the information the LR parser has when an error is detected, which it can use to communicate with the user about the error. Of course, just announcing there is a "syntax error," "parser error," or "stack overflow" is not very helpful for the user because it

does not help then find and fix errors. LR uses the parse state and the current input token to decide that there is an error [46].

Many error recovery systems were designed to generate better error messages such as panic mode, minimum distance, error productions, or empty table slots. Error recovery systems attempt to detect all syntax errors in the input by continuing parsing after error discovery, and they try to avoid spurious error messages. The spurious error messages are not real errors in the input but result from the continuation of the parser after an error is detected. However, error correction systems were also designed because a parser with an error recovery method can no longer deliver a parse tree if the input contains errors. Error correction systems transform input into syntactically correct input to be able to generate a parse tree with the associated semantic actions. Error correction systems commonly change the input by deleting, inserting, or changing symbols. [45].

The parser generated by Bison has a Look Ahead Correction. This mechanism suspends the normal parser whenever it fetches a new token from the scanner, runs an exploratory parser, and uses a temporary state stack. When the exploratory parse reaches a shift action, normal parsing is resumed. If the exploratory parse reaches an error, the parser announces a syntax error. When the compiler writer enables verbose syntax error messages, the parser must discover the list of expected tokens and perform a separate exploratory parse for each token in the grammar. However, in a consistent parser state with a default reduction, no lookahead is needed to determine the following parser action, so the parser will not attempt to fetch a token from the scanner [47].

An efficient and automatic way of generating error messages is using the Merr (Meta Error Generator) tool, which was mentioned in Chapter 2. Merr, or its underlying concept, is used by many other syntax analysis tools, such as the Menhir parser generator used for OCaml language and iYacc, a parser generator for Unicon language. This research also uses Merr. Merr uses parse states and current input tokens to generate an error reporting function and allow the compiler designer to craft customized error messages for each State in the parser automaton. Moreover, it enables writing different error messages for each erroneous input token for the same State [48].

Why do we still need better syntax error messages despite all these techniques? Chapter 4 shows cases where the state of art compilers such as GNU GCC (GCC) and Microsoft Visual C++ Compiler (MSVC) report common and simple syntax errors with misleading error messages. To explain some of the limitations of the LR parser to report syntax errors, let us examine how the LALR(1) parser (Let us call it ToyC) of the grammar in Figure 5.4 reports the error in the program in Figure 5.5 which is similar to the examples that were discussed in Chapter 4. Because the grammar for C++ is giant, a

grammar with a small number of production rules is shown. However, drawing the parsing table or the automaton on one page is hard, even for such a small grammar. Following the LR algorithm that is presented in the previous section, the ToyC's configuration at the point of reaching token "}" at line 6 is as the following:

Step N: the stack has

State 0	DeclarationList	State 4	FunctionHeader	State 9	{	State 16	OptStatementList	State 34
---------	-----------------	---------	----------------	---------	---	----------	------------------	----------

The action is shift } and enter State 56:

Step N+1: the stack has

State 0	DeclarationList	State 4	FunctionHeader	State 9	{	State 16	OptStatementList	State 34	}	State 56
---------	-----------------	---------	----------------	---------	---	----------	------------------	----------	---	----------

The next token is IDENTIFIER, the action is reduce by the rule

`BlockStatement` → '{' `OptStatementList` '}'

The length of right of the rule side is 3, so the parser pops 6 elements from the stack, pushes `BlockStatement`, and enters State 17

Step N+2: the stack has

State 0	DeclarationList	State 4	FunctionHeader	State 9	Block Statement	State 17
---------	-----------------	---------	----------------	---------	-----------------	----------

The action is reduce by the rule

`FunctionDefinition` → `FunctionHeader` `BlockStatement`

The length of the right side of the rule is 2, so the parser pops 4 elements, pushes `FunctionDefinition`, and enters the State 6.

Step N+3: the stack has

State 0	DeclarationList	State 4	FunctionDefinition	State 6
---------	-----------------	---------	--------------------	---------

The action is reduce by the rule

`Declaration` → `FunctionDefinition`

The parser pops 2, pushes `Declaration`, and enters State 14

Step N+4: the stack has

<i>State 0</i>	DeclarationList	<i>State 4</i>	Declaration	<i>State 14</i>
----------------	-----------------	----------------	-------------	-----------------

The action is reduce by the rule

`DeclarationList → Declaration`

The parser pops 2, pushes DeclarationList, and enters State 4

Step N+5: the stack has

<i>State 0</i>	DeclarationList	<i>State 4</i>
----------------	-----------------	----------------

The action is reduce by the rule

`GlobRegion → DeclarationList`

The parser pops 2, pushes DeclarationList, and enters State 3

Step N+6: the stack has

<i>State 0</i>	GlobRegion	<i>State 3</i>
----------------	------------	----------------

The action is reduce by the rule

`Program → GlobRegion`

The parser pops 2, pushes DeclarationList, and enters State 2

Step N+7: the stack has

<i>State 0</i>	Program	<i>State 2</i>
----------------	---------	----------------

The State at the top of the stack is State 2, and the current input token is IDENTIFIER. The parser finds no action related to IDENTIFIER, so it announces an error. Figure 5.3 and Figure 5.4 show State 0 and State 2.

The information available at this point does not help generate clear and helpful error messages. This is in agreement with the analysis of error messages of GCC and MSVC in Chapter Four, which showed that compiler error messages are often unclear and/or unhelpful. For example, the GCC error message in Figure 5.6 for the program in Figure 5.5 agrees with the idea that error messages are unhelpful. At the global level of the program, the parser does not expect the expressions and

statements level of the programs. There are no rules or corresponding automaton states to recognize them or have allow those tokens in those parse.

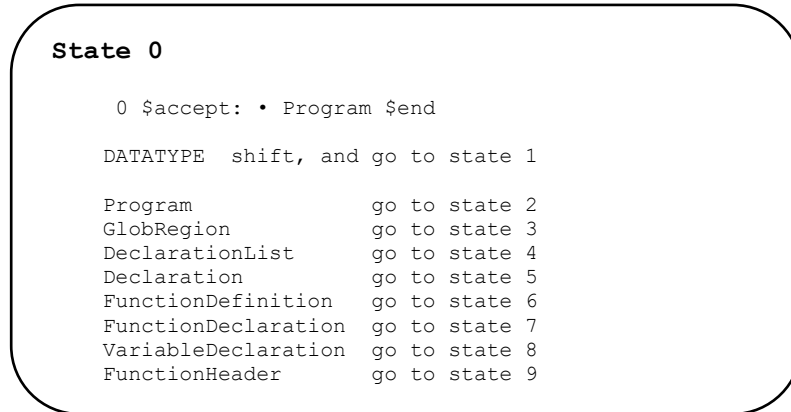


Figure 5.3 State 0 of the automaton of the grammar in Figure 5.1

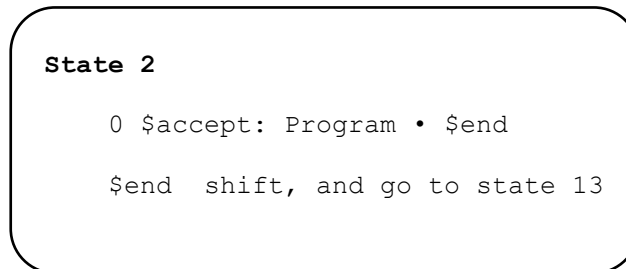


Figure 5.4 State 2 of the automaton of the grammar in Figure 5.1

```

%token IDENTIFIER DATATYPE NUMBER RETURN WHILE ASSIGN LOP
%%
Program:GlobRegion
GlobRegion:DeclarationList
DeclarationList:Declaration
|DeclarationList Declaration
Declaration:FunctionDeclaration
|VariableDeclaration
|FunctionDefinition
FunctionDefinition: FunctionHeader BlockStatement
FunctionDeclaration: FunctionHeader ';'
VariableDeclaration:DATATYPE ParameterList ';'
FunctionHeader: DATATYPE Identifier '(' OptParamaterList ')'
OptParamaterList:ParameterList
|/*empty*/
ParameterList:Identifier
|ParameterList ',' Identifier
StatementList: Statement
|StatementList Statement

```

```

Statement: ExpressionStatement
|AssignmentStatement
|VariableDeclaration
|EmptyStatement
|BlockStatement
|WhileStatement
|ReturnStatement
BlockStatement: '{' OptStatementList '}'
WhileStatement: WHILE '(' Expression ')' Statement
EmptyStatement: ';'
ReturnStatement: RETURN OptExpression ';'
OptStatementList: StatementList
/*empty*/
OptExpression: Expression
| /*empty*/
ExpressionStatement: Expression ';'
Expression: LogicalExpression
LogicalExpression: LogicalExpression LOP AdditiveExpression
|AdditiveExpression
AssignmentStatement: Identifier ASSIGN ExpressionStatement;
AdditiveExpression: AdditiveExpression '+' MultiplicativeExpression
|AdditiveExpression '-' MultiplicativeExpression
|MultiplicativeExpression
MultiplicativeExpression: MultiplicativeExpression '*' PrimaryExpression
MultiplicativeExpression '/' PrimaryExpression
|PrimaryExpression
PrimaryExpression: Literal
|Identifier
|ParenthesizedExpression
Literal: NumericLiteral
NumericLiteral: NUMBER
ParenthesizedExpression: '(' Expression ')'
Identifier: IDENTIFIER
%%

```

Figure 5.4 A grammar for ToyC.

```

1  int x, y, z;
2  int main()
3  {
4      while(x<y)
5          z=z+1;
6  }
7  z=x+y;
8  }

```

Figure 5.5 An Erroneous program for the language of the grammar in Figure 5.4

```

$ g++ prog1.cpp
prog1.cpp:7:1: error: 'z' does not name a type
  7 | z=x+y;
    | ^
prog1.cpp:8:1: error: expected declaration before '}' token
  8 | }
    | ^
$ 

```

Figure 5.6 GCC's error message for the program in Figure 5.5

To get a closer look, let us compare the parser configuration for the program in Figure 5.5 and the parser configuration for program 2 in Figure 5.7 when the error was detected. For example, Figure 5.5 shows State 34, that when the error is detected for program 2 :

```

State 34

  25 BlockStatement: '{' OptStatementList • '}'
      '}' shift, and go to state 56

```

Figure 5.5 State 34 of the automaton of the grammar in Figure 5.1

In this case, knowing the state number is helpful. The expected token is "}" while the current token is \$end. Also, this agrees with the GCC's error message for program 2, as shown in Figure 5.8. It is a helpful error message.

```

1  int x, y, z;
2  int main()
3  {
4      while(x<y){
5          | z=z+1;
6          }
7          z=x+y;
8

```

Figure 5.7 Erroneous program.


```

$ g++ program2.cpp
program2.cpp: In function 'int main()':
program2.cpp:7:9: error: expected '}' at end of input
   7 |     z=x+y;
     |           ^
program2.cpp:3:1: note: to match this '{'
   3 | {
     | ^
$ █

```

Figure 5.8 GCC's error message for the program in Figure 5-7

Summarizing the available information from the LR parser or the previously mentioned recovery or Merr tool are the state number, the current token, and the expected legal tokens available from the State where the error is detected. All this information is insufficient to tell the user that the error is outside the function boundaries. So, the previous examples show the limitation of the parser in reporting some cases of detected errors. The proposed solution for this problem aims to overcome the parser's limitations in these cases.

5.5 New solution: 3-phase parsing techniques.

From studying the parser behavior in detecting syntax errors in the previous section, analysis of the mainstream compilers in Chapter 4, and the available information provided by the Merr tool, we attack the problem with the following strategy:

1. Divide and Conquer: minimize the number of states by focusing on the subset of the production rule that governs the validation of the skeleton of the large component of the input source code. Moreover, separating the rules for the outer skeleton from those for the inner skeleton will give us more advantages in customizing error messages. The number of states in a giant grammar like the C++ grammar is in the thousands. Using LALR, which merges similar states, only helps a little because it increases the number of expected input tokens in each state.
2. The proposed solution uses the lexical analysis to customize the error messages and ignore some erroneous input.
3. Imitate how the teachers of introductory programming courses correct the erroneous programs with their students.

As a result, the proposed technique is to parse the source code in three phases. Phase one analyzes the source code into its major constituent parts: functions. Phase two analyzes control structures inside the functions' bodies in phase one. Finally, phase three analyzes the fine-grained statements and expressions of the source code. Each phase does lexical analysis and syntax analysis and generates abstract syntax trees. Also, it uses Merr [48] with each phase's parser to automatically produce a mapping of parse states to diagnostic messages, so each parser has its own error reporting function.

Phase one

Phase one analyzes the outer skeletal structure of the functions in the source code. Phase one decides whether the source code conforms to function declaration and function definition rules. The lexical analysis of this phase recognizes only the tokens related to the function declaration and definition. This phase's syntax analysis validates the rules related to the function declaration and definition. It reports errors using a function `yyerror_a()` that is generated by the Merr tool and generates an abstract syntax tree (AST-a). The AST-a has a branch for each function in the source code that contains notations such as the function name, the return type, the location, and pointers to children. In phase one, the first child of a function is the header part, and the second is the body part. These children are of type string and are not validated at this phase for any rules. The next phase analyzes the second child's body and decides if it follows the rules for the control structures.

Phase two

Phase two analyzes the functions' bodies that are generated from phase one. Phase two decides whether function bodies conform to the rules for all kinds of loops and conditional control structure rules. The lexical analysis of this phase recognizes only the tokens related to the iterations, `if`-statement, and switch structures and validates the rules related to the iteration, `if`-statement, and switch structures. It reports errors using the function `yyerror_b()` generated by the Merr tool and generates an abstract syntax tree (AST-b). The AST-b has a node for each control structure that holds information about the location and pointers to the children. The first child is the header part, and the second child is the body part. The children are of type string, and they are not validated at this phase for any rules. The next phase analyzes the header and body parts processed in phases one and two.

Phase three

Phase three analyzes the statements found in the functions' headers and bodies, control structures' headers and bodies, and those found outside the function boundaries. These statements are aggregated

into one string buffer and are notated with their parents' information. Finally, it reports errors using the Merr tool, has its `yyerror_d()`, and generates an abstract syntax tree (AST-d).

5.6 Implementation of the 3-phase parsing techniques in an Educationally Customized

Compiler:

A compiler prototype for C++ for the introductory level was developed. The prototype is developed for phases one and two, and the code is written to make both parsers work together. The following sections present the grammar and the lexical analysis rules. However, the rest of the code is presented in Appendix B, which includes `yyerror.c` and `meta.err` for parser 1, `berror.c`, `meta.err` for the parser 2, and the `main.c`.

Parser 1 Grammar

```
%{
    int yylex();
    int _yyerror(char *, int);
#include<string.h>
#include "yyerror.h"
#include "p.h"
char gfuncName[50];
int glastLine;
extern struct tree *root;
struct location loc={-1,-1,-1,-1};
extern int yylineno;
extern char* mybuff;
int _initial_preprocess=0, _initial_rawtxt=0, _initial_out_rawtxt=0, _initial_para=0,
_initial_curlybracket=0, _initial_rawtxt_body=0;
}%
#define parse.trace
%locations
%token <s> OPENPARA CLOSEPARA OPENB CLOSEB
%token <s> VAR DATATYPE
%token <s> RAWTXT
%token <s> NUM
%token <s> KEY
%token <s> SEMICOLON
%token <s> LSTRING
%token <s> LT LT2 CIN COUT
%token <s> GT GT2
%token <s> HASH INCLUDE USING DEFINE
%union {
    char *s;
    struct tree *n;
}
%type <n> file program component funcndef funcndec funcheader funcbody
%type <s> rawtxt out_rawtxt curlybracket para unbalanced_para preprocess rawtxt_header_seq
rawtxt_body_seq

%%
file: program {$$=newTree("file 1",loc,1,$1);root=$$;}
```

```

;
program: component {$$=newTree("program 2",loc,1,$1);}
      | program component{$$=newTree("program 2",loc,2,$1,$2);}
;
component: funcundef {$$=newTree("c_fdef 7",loc, 1,$1);}
      |out_rawtxt {$$=newTree("c_raw 7",loc, 1,$1);}
      |funcundef{$$=newTree("c_raw 7",loc, 1,$1);}
;

funcundef: funcheader SEMICOLON {/*$$=newTree("funcundef 4",loc, 2,$1, newLeaf("funcundef",$2));*/}
;
out_rawtxt:DATATYPE VAR SEMICOLON {$$=NULL;}
      |preprocess{$$=NULL;}
;

preprocess:HASH INCLUDE {$$=NULL;}
      |HASH DEFINE {$$=NULL;}
      |USING {$$=NULL;}
;

funcundef:funcheader funcbody {$$=newTree("funcdef 3",loc, 2,$1,$2);}
;

funcheader:DATATYPE VAR OPENPARA rawtxt_header_seq CLOSEPARA {$$=newTree("funcheader 5",loc,
5,newLeaf("funcheader",$1),newLeaf("funcheader",$2),
newLeaf("funcheader",$3),$4,newLeaf("funcheader",$5));strncpy(gfuncName,$2,50);}
;
funcbody:OPENB rawtxt_body_seq CLOSEB
      {
          loc.first_column=@1.first_column;
          loc.first_line=@1.first_line;
          loc.last_column=@3.last_column;
          loc.last_line=@3.last_line;
          glastLine=@3.last_line;
          $$=newTree("funcbody 6",loc, 3,newLeaf("funcbody",$1),$2,newLeaf("funcbody",$3));}
;

rawtxt_header_seq:%empty {$$=NULL;}
      |rawtxt {$$=NULL;}
      |para {$$=NULL;}
      |rawtxt_header_seq rawtxt {$$=NULL;}
      |rawtxt_header_seq para {$$=NULL;}
;

curlybracket:OPENB rawtxt_body_seq CLOSEB {$$=NULL;}
      |OPENB CLOSEB {$$=NULL;}
;

para:OPENPARA rawtxt_header_seq CLOSEPARA {$$=NULL;}
      |OPENPARA CLOSEPARA {$$=NULL;}
;

rawtxt_body_seq: %empty{$$=NULL;}
      |rawtxt {$$=NULL;}
      |curlybracket {$$=NULL;}
      |unbalanced_para {$$=NULL;}
      |rawtxt_body_seq rawtxt {$$=NULL;}

```

```

        |rawtxt_body_seq curlybracket {$$=NULL;}
        |rawtxt_body_seq unbalanced_para {$$=NULL;}
;
unbalanced_para:OPENPARA {$$=NULL;}
        |CLOSEPARA {$$=NULL;}
;
rawtxt:KEY {$$=NULL;}/*sprintf(mybuff,"%s",$2);$$=mybuff;*/
        |CIN {$$=NULL;}
        |COUT {$$=NULL;}
        |VAR {$$=NULL;}
        |DATATYPE {$$=NULL;}
        |LSTRING {$$=NULL;}
        |SEMICOLON {$$=NULL;}
        |RAWTXT {$$=NULL;}
        |NUM {$$=NULL;}
        |GT {$$=NULL;}
        |LT {$$=NULL;}
        |GT2 {$$=NULL;}
        |LT2 {$$=NULL;};

```

Parser 1 Lexical Specification

```

%option yylineno

%option noyywrap
float ([0-9]*\.[0-9]+)|([0-9]+\.)
exponent [eE][+]?[0-9]+
%{

#include <stdio.h>
#include "gram.tab.h"

#define YY_USER_ACTION \
    yylloc.first_line=yylloc.last_line;\
    yylloc.first_column=yylloc.last_column;\
    for(int i= 0; yytext[i]!='\0'; i++){
        if(yytext[i]=='\n') {\
            yylloc.last_line++;
            yylloc.last_column=0;
        }
        else{\
            yylloc.last_column++; } }
    }

%%
"/*"([^*|"*+[^*/])*"*+ "/" {}
"//".*"\n" {}
[\\t\\f\\v\\r\\n ]+ { /* Ignore whitespace. */ }
 "(" {yylval.s=strdup(yytext);return OPENPARA; }
 ")" {yylval.s=strdup(yytext);return CLOSEPARA;}
 "{" {yylval.s=strdup(yytext);return OPENB;}
 "}" {yylval.s=strdup(yytext);return CLOSEB;}
 ";" {yylval.s=strdup(yytext);return SEMICOLON;}
 ">" {yylval.s=strdup(yytext);return GT;}
 "<" {yylval.s=strdup(yytext);return LT;}
 ">>" {yylval.s=strdup(yytext);return GT2;}
 "<<" {yylval.s=strdup(yytext);return LT2;}
 "char" {yylval.s=strdup(yytext);return DATATYPE;}

```

```

"int" {yylval.s=strdup(yytext);return DATATYPE;}
"float" {yylval.s=strdup(yytext);return DATATYPE;}
"double" {yylval.s=strdup(yytext);return DATATYPE;}
"void" {yylval.s=strdup(yytext);return DATATYPE;}
"string" {yylval.s=strdup(yytext);return DATATYPE;}
"while" {yylval.s=strdup(yytext);return KEY;}
"do" {yylval.s=strdup(yytext);return KEY;}
"switch" {yylval.s=strdup(yytext);return KEY;}
"case" {yylval.s=strdup(yytext);return KEY;}
"if" {yylval.s=strdup(yytext);return KEY;}
"for" {yylval.s=strdup(yytext);return KEY;}
"return" {yylval.s=strdup(yytext);return KEY;}
"cin" {yylval.s=strdup(yytext);return CIN;}
"cout" {yylval.s=strdup(yytext);return COUT;}
"#" {yylval.s=strdup(yytext);return HASH;}
#include.*"\n" {yylval.s=strdup(yytext);return INCLUDE;}
using.*"\n" {yylval.s=strdup(yytext);return USING;}
#define" {yylval.s=strdup(yytext);return DEFINE;}

[a-zA-Z_][a-zA-Z_0-9]* {yylval.s=strdup(yytext); return VAR;}
"0"[xX][0-9a-fA-F]+ {yylval.s=strdup(yytext); return NUM;}
"0"[0-7]+ {yylval.s=strdup(yytext); return NUM;}
[0-9]+ {yylval.s=strdup(yytext); return NUM;}
{float}{exponent}? {yylval.s=strdup(yytext); return NUM;}
[0-9]+{exponent}? {yylval.s=strdup(yytext); return NUM;}
"\\"(\\"|"[^"]")*" {yylval.s=strdup(yytext); return LSTRING;}

. /* fprintf(stderr, "lexical error: %d\n", yytext[0]);*/yylval.s=strdup(yytext);return
RAWTXT;}

```

Parser 2 Grammar

```

%{
    int blex(void);
    int _berror(char *s, int);

#define BDEBUG 1;
#include "berror.h"
#include "p.h"
extern struct tree *b_root;
struct location b_loc={-1,-1,-1,-1};
%}
#define parse.trace

%token <s> WHILE FOR DO IF ELSE SWITCH
%token <s> OPENPARA CLOSEPARA OPENB CLOSEB RAWTXT
%token <s> COLON SEMICOLON CASE DEFAULT
%token <s> OP INC DECL STREAM OBJ STAT PRE NUM STRING VAR HASH

%union {
    char *s;
    struct tree *n;
}

%%
func_body:statement_seq
;

statement:iteration_statement
         |selection_statement
         |compound_statement
         |rawtxt_seq
;

compound_statement:OPENB statement_seq_opt CLOSEB
;

statement_seq_opt:%empty
                 |statement_seq
;

statement_seq:statement
              |statement_seq statement
;

iteration_statement:WHILE OPENPARA rawtxt_seq1 CLOSEPARA statement_seq
                  |DO OPENB statement_seq CLOSEB WHILE OPENPARA rawtxt_seq1 CLOSEPARA SEMICOLON
                  |FOR OPENPARA rawtxt_seq1 SEMICOLON rawtxt_seq1 SEMICOLON rawtxt_seq1 CLOSEPARA statement_seq
;

selection_statement:IF OPENPARA rawtxt_seq1 CLOSEPARA statement_seq_opt
                   |IF OPENPARA rawtxt_seq1 CLOSEPARA statement_seq_opt ELSE statement_seq_opt
                   |SWITCH OPENPARA rawtxt_seq1 CLOSEPARA OPENB case_stmt_seq CLOSEB
;

case_stmt_seq:case_stmt_seq case_stmt

```

```

        |case_stmt
;

case_stmt:CASE rawtxt_seq COLON statement_seq_opt
        | DEFAULT rawtxt COLON statement_seq_opt

rawtxt_seq:rawtxt_seq1
        |rawtxt_seq1 SEMICOLON
;

rawtxt_seq1: stat
        |stat rawtxt1
        |exp_seq
        |exp_seq rawtxt1
;

rawtxt1:rawtxt
        |rawtxt1 rawtxt
;

rawtxt:RAWTXT
        |rawtxt RAWTXT
        |STREAM
        |rawtxt STREAM
        |NUM
        |STRING
;

exp_seq: exp
        |exp_seq OP
        |exp_seq OP exp
;

exp:VAR
        |NUM
        |STRING
        |VAR OPENPARA exp_seq CLOSEPARA
        |VAR OPENPARA CLOSEPARA
        |OPENPARA exp_seq CLOSEPARA
;

stat:STAT
        |DECL
        |OBJ
        |INC VAR
        |VAR INC
;

```

Parser 2 Lexical Specification

```

%option yylineno
%option noyywrap
float ([0-9]*\.[0-9]+)|([0-9]+\.)
exponent [eE][+]?[0-9]+

%{
#include <stdio.h>
#include "gramb.tab.h"
#define YY_DECL extern int blex(void)
}%

%%
"/" * ([^*] | "*" + [^*/]) "*" + "/" {}

```



```

"//".*"\\n" {}
[\\t\\f\\v\\r\\n]+ { /* Ignore whitespace. */ }
"(" {return OPENPARA;}
")" {return CLOSEPARA;}
"{" {return OPENB;}
"}" {return CLOSEB;}
"*" {return OP;}
"+" {return OP;}
"++" {return INC;}
"--" {return OP;}
"--" {return INC;}
"|" {return OP;}
%" {return OP;}
"&" {return OP;}
"||" {return OP;}
"&&" {return OP;}
"/" {return OP;}
"!" {return OP;}
"=" {return OP;}
"!=" {return OP;}
"==" {return OP;}

"while" {return WHILE;}
"for" {return FOR;}
"do" {return DO;}
"if" {return IF;}
"switch" {return SWITCH;}
"else" {return ELSE;}
"case" {return CASE;}
"default" {return DEFAULT;}
":" {return COLON;}
";" {return SEMICOLON;}

">" {blval.s=strdup(yytext);return OP;}
"<" {blval.s=strdup(yytext);return OP;}
">=" {blval.s=strdup(yytext);return OP;}
"<=" {blval.s=strdup(yytext);return OP;}
">>" {blval.s=strdup(yytext);return STREAM;}
"<<" {blval.s=strdup(yytext);return STREAM;}
"char" {blval.s=strdup(yytext);return DECL;}
"int" {blval.s=strdup(yytext);return DECL;}
"float" {blval.s=strdup(yytext);return DECL;}
"double" {blval.s=strdup(yytext);return DECL;}
"void" {blval.s=strdup(yytext);return DECL;}
"return" {blval.s=strdup(yytext);return STAT;}
"cin" {blval.s=strdup(yytext);return OBJ;}
"cout" {blval.s=strdup(yytext);return OBJ;}
"#" {blval.s=strdup(yytext);return HASH;}
#include".*"\\n" {blval.s=strdup(yytext);return PRE;}
using".*"\\n" {blval.s=strdup(yytext);return PRE;}
define" {blval.s=strdup(yytext);return PRE;}

[a-zA-Z_][a-zA-Z_0-9]* {blval.s=strdup(yytext); return VAR;}
"0"[xX][0-9a-fA-F]+ {blval.s=strdup(yytext); return NUM;}
"0"[0-7]+ {blval.s=strdup(yytext); return NUM;}
[0-9]+ {blval.s=strdup(yytext); return NUM;}

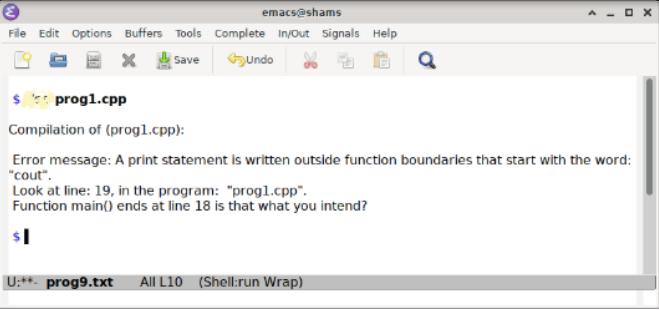
```

```
{float}{exponent}? {blval.s=strdup(yytext); return NUM;}
[0-9]{exponent}? {blval.s=strdup(yytext); return NUM;}
"\"(\\"|^[^\\"])*\" {blval.s=strdup(yytext); return STRING;}
. {return RAWTXT;}
```

5.7 Sample error messages generated by EduCC

This section shows a typical error message for the associated code. Note that the error message clearly gives more friendly content, some code associated with the error, and explanations.

Message 1, function boundaries



Assume that the actual error in the code is that the user added a close curly bracket at the end of line 18.

```

1  /* The fortune Teller -
2     * a simple program introducing some
3     * fundamental programming concepts.
4     */
5  #include<iostream> // include a library
6  using namespace std;
7  int main() // main() starts the program
8  {
9     // ----- Variable Declarations -----
10     int favorite; // create a variable to store the favorite number
11     int disliked; // create a variable to store the disliked number
12     int lucky; // create a variable to store the lucky number
13     // ----- Get user input -----
14     cout << "Enter your favorite number (no decimals): "; // output
15     cin >> favorite; // user input
16     cout << "Enter a number you don't like (no decimals): ";
17     cin >> disliked;
18     lucky = ((abs(favorite-disliked))*13) % 10;
19     cout << endl << "Your secret, lucky number is: " << lucky << endl;
20     if(lucky < 0){ // conditional, clause less than 0
21         cout << "Try to be less negative." << endl;
22     }
23     if(lucky >= 0 && lucky < 5){ // 0 to 4 inclusive
24         cout << "Think bigger!" << endl;
25     }
26     if(lucky >= 5 && lucky < 9){ // 5 to 8 inclusive
27         cout << "Today you should embrace technology." << endl;
28     }
29     if(lucky == 9){ // exactly 9
30         cout << "Today is your lucky day!" << endl;
31     // ----- Code to help the program exit "gracefully" -----
32     cin.ignore();
33     cout << "Press enter to quit." << endl;
34     cin.ignore();
35     return 0;
36 }

```

Message 2, function boundaries

emacs@shams

File Edit Options Buffers Tools Complete In/Out Signals Help

Save Undo

\$ **prog2.cpp**

Compilation of (prog2.cpp):

Error message: "{" or ";" is expected; if this is a function definition a "{" is required, but if this is a function declaration a ";" is required before the word/character: "(".

Look at line: 37, in the program: "prog2.cpp".

Function area() ends at line 35 is that what you intend?

\$ |

Assume the actual error in the code is that the user wrote open parenthesis "(" instead of open curly bracket "{" in line 37.

```

1      #include<iostream>
2      using namespace std;
3
4      // the following code create a functions to calculate the
5      // area of a circle, it overloads the same function
6      // first function area
7      // takes a double as an argument, which
8      // represents the radius of a circle
9      // and returns the area of a circle
10
11     // second and third area functions
12     // take the radius and the area as
13     // arguments and sets area to be
14     // correct without returning a value
15
16     void area(double,double);
17     void area(double,double*);
18     double area(double);
19
20     int main(){
21         double a;
22         a = area(3);
23         cout << a << endl;
24         double r2 = 4, myarea = 78;
25         area(r2, myarea);
26         cout << "the area of a circle ";
27         cout << "with radius: " << r2;
28         cout << " is " << myarea << endl;
29         double area2 = 23;
30
31         double *p = &area2;
32         area(5,p); // pointer function
33         // area(5,&area2); // pointer function
34         cout << area2 << endl;
35     }
36
37     void area(double x, double *area)
38     {
39         cout << "pointer\n";
40         *area = 3.1417 * x * x;
41     }
42
43     void area(double x, double &area){
44         cout << "pass by reference\n";
45         area = 3.1417 * x * x;
46     }
47
48     double area(double x){
49         cout << "pass by value with a return\n";
50         double answer;
51         answer = 3.1417 * x * x;
52         return answer;
53     }

```

Message 3, function boundaries

Compilation of (prog3.cpp):

Error message: A statement is written outside function boundaries that start with the word: "if". Look at line: 61, in the program: "prog3.cpp". Function room2() ends at line 60 is that what you intend?

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66

```

```

#include<iostream>
using namespace std;

// Function prototypes
int room1(); // addition
int room2(); // subtraction
//int goldstars = 0; // global variable, can be 'seen' in all functions.

int main(){
    int current_room = 1; // local variable, only exists in main()
    cout << "This is a math, adventure game.\n";
    do{
        cout << endl << endl;
        cout << "current_room: " << current_room << endl;
        cout << "Gold stars: " << goldstars << endl;

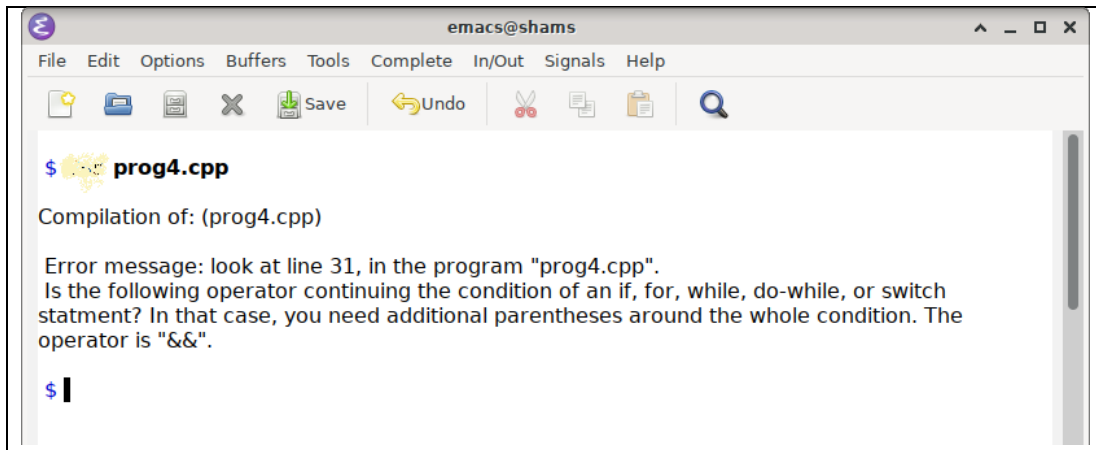
        switch(current_room){
            case 1:
                current_room = room1();
                break;
            case 2:
                current_room = room2();
                break;
            default:
                cout << "Error: " << current_room << endl;
        }
    }while(current_room != 3);
    cout << "You win, Bye!\n";
}

// function definitions (code)
int room1(){
    static int visited = 0; // keeps track of whether this room has been visited already
    int choice;
    if(visited == 0){ // if not visited, get a gold star
        goldstars = goldstars + 1;
        visited = 1; // change the value so next time it counts as visited
    }
    cout << "This is room 1.\n";
    cout << "1) stay here\n";
    cout << "2) go to the subtraction room.\n";
    cin >> choice;
    if (choice == 1){
        return 1; // 'goto' room 1
    }
    if (choice == 2){
        return 2; // goto room 2
    }
}

int room2(){
    int choice;
    cout << "The subtraction room (2).\n";
    cout << "1) stay\n";
    cout << "2) room 1\n";
    cout << "3) exit\n";
    cin >> choice;
    if(choice == 1)
        return 2; // stay in room 2
    if(choice == 2){
        return 1; // go to room 1
    }
    if(choice == 3){
        return 3; // will exit the game
    }
}

```

Message 4, control structure header



Assume that the actual error in the code is that the user missed the parentheses of the if statement header in line 31.

```

1      #include<iostream>
2      using namespace std;
3
4      int main(){
5          double value1, value2, result ,count=1;
6          int choice;
7          cout << "This is a simple calculator program.\n";
8          cout << "Enter two values and select the operation ";
9          cout << "you want it to perform.\n";
10         cout << "Value 1: ";
11         cin >> value1;
12         cout << "Value 2: ";
13         cin >> value2;
14         cout << "Pick your operation:\n";
15         // note the use of \n on the next line to create a menu.
16         cout << "1) Addition \n2) Subtraction \n3) Multiplication \n4) Division\n";
17         cin >> choice;
18         ++count;
19         if(choice == 1){
20             result=value1+value2;
21             cout << value1 << " + " << value2 << " = " << result << endl;
22         }
23         if(choice == 2){
24             result=value1-value2;
25             cout << value1 << " - " << value2 << " = " << result << endl;
26         }
27         if(choice == 3){
28             result=value1*value2;
29             cout << value1 << " * " << value2 << " = " << result << endl;
30         }
31         if (choice == 4)&&(value2 != 0) { // it should be : if (choice == 4)&&(value2 != 0){
32             result=value1/value2;
33             cout << value1 << " / " << value2 << " = " << result << endl;
34         }
35     }

```

Message 5, control structure header

emacs@shams

File Edit Options Buffers Tools Complete In/Out Signals Help

Save Undo

\$ prog5.cpp

Compilation of: (prog5.cpp)

Error message: look at line 35, in the program "prog5.cpp".
Semicolons is required in the for header, expected two semicolons inside the for's parentheses (i.e for(int x=1;x<10;x++). You may fix that by adding two ';' in the proper places: for (statement; condition ; statment) before ").

\$ |

Assume that the actual error in the code is that the user missed the semicolons of the for statement header in line 35, and the variable "i" was not declared.

```

1      #include<iostream>
2      #include<cstdlib>
3      #include<ctime>
4      using namespace std;
5
6      // get_num() asks the user for a
7      // positive number and returns it
8      int get_num();
9      // roll() takes an integer as an
10     // argument, rolls that many dice
11     // randomly and returns the sum
12     int roll(int);
13
14     int main(){
15         int num_dice,sum;
16         srand(time(0));
17         // get number of dice from user
18         num_dice = get_num();
19         // roll dice
20         sum = roll(num_dice);
21         // print sum
22         cout << "Rolled a: " << sum << endl;
23         return 0;
24     }
25
26     int get_num(){
27         int choice = 0;
28         cout << "please enter a positive value: ";
29         cin >> choice;
30         return choice;
31     }
32
33     int roll(int n){
34         int die = 0;
35         for(i < n){ // assume it should be for (int i=1 ; i<n ;
36             die = die + (rand() % 6)+1;
37             cout << i << ";<<die << " ";
38         }
39         cout << endl;
40         return die;
41     }

```

Message 6, control structure header

emacs@shams

File Edit Options Buffers Tools Complete In/Out Signals Help

Save Undo

\$ **prog6.cpp**

Compilation of: (prog6.cpp)

Error message: look at line 35, in the program "prog6.cpp".
Do you mean open bracket '{' ? because usually for statement start with '{', or any other correct statement before: "i".

\$ |

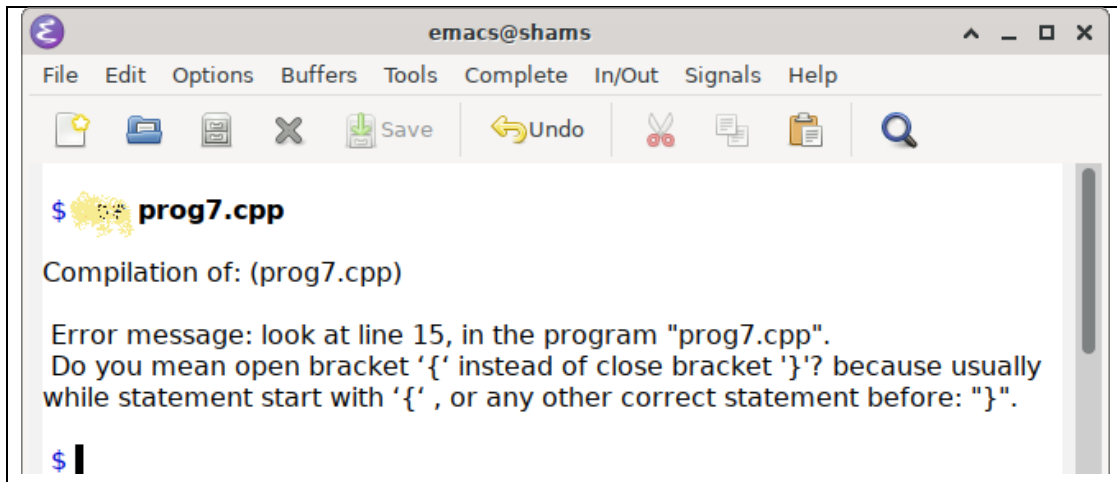
Assume that the actual error in the code is that the user missed the semicolons of the for statement header in line 35, and the variable "i" was not declared.

```

1  #include<iostream>
2  #include<cstdlib>
3  #include<ctime>
4  using namespace std;
5
6  // get_num() asks the user for a
7  // positive number and returns it
8  int get_num();
9  // roll() takes an integer as an
10 // argument, rolls that many dice
11 // randomly and returns the sum
12 int roll(int);
13
14 int main()
15 {
16     int num_dice,sum;
17     srand(time(0));
18     // get number of dice from user
19     num_dice = get_num();
20     // roll dice
21     sum = roll(num_dice);
22     // print sum
23     cout << "Rolled a: " << sum << endl;
24     return 0;
25 }
26
27 int get_num(){
28     int choice = 0;
29     cout << "please enter a positive value: ";
30     cin >> choice;
31     return choice;
32 }
33
34 int roll(int n){
35     int die = 0;
36     for( i = 0, i < n, i++) // assume that it should be for( int i; i < n; i++)
37     {
38         cout << i << " "; << die << " ";
39         die = die + (rand() % 6)+1;
40     }
41     cout << endl;
42     return die;
43 }

```


Message 7, control structure body



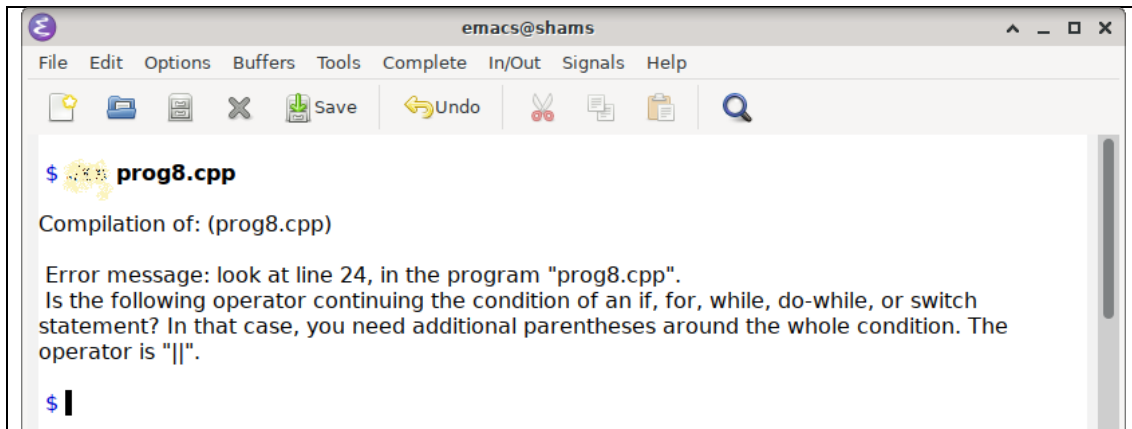
Assume that the actual error is that the user used close curly '}' bracket instead of open curly bracket '{' in line 15.

```

1      #include<iostream>
2      #include<cstdlib>
3      using namespace std;
4
5      int main(){
6          // variables
7          int computerguess;
8          int HorL=1;
9          // instructions to user
10         {cout << "Pick a number (1-100) for the computer to guess.\n";
11         cout << "Don't forget the number.\n";
12         // 1) computer's first guess
13         computerguess = 1 + rand() % 100;
14         while(HorL!=3)
15             |
16             cout << "I guessed: " << computerguess << endl;
17             cout <<"(1)higher or (2)lower or (3)correct?\n";
18             cin >> HorL;
19             // user says higher/lower or correct
20             if(HorL == 1){
21                 computerguess = computerguess + 1;
22             }
23             if(HorL == 2){
24                 computerguess = computerguess - 1;
25             }
26             if(HorL == 3){
27                 cout << "Yay, I was right. Computers rock!\n";
28             }
29             // computer guesses again
30             // loop to 1 if not correct
31             // message
32         }

```

Message 8, control structure body



Assume that the actual error is that the user add extra close parenthesis in *while* part of the *do-while* statement in line 24.

```

1      /* The game of NIM
2      */
3      #include<iostream> // include two libraries
4      #include<cstdlib>
5      using namespace std;
6      int main() // main() starts the actual program
7      {
8          // ----- Variable declarations -----
9          int num_objects = 23;
10         int current_player = 1;
11         int move;
12         // ----- Beginning of the main game loop -----
13         do {
14             if (current_player == 1) { // conditional: if
15                 cout << "Player 1 enter your move (1-3): "; // output
16                 cin >> move; // input
17                 while ((move < 1) || (move > 3) || (move > num_objects)){
18                     cout << "Illegal move. \nEnter a new move (1-3): ";
19                     cin >> move;
20                 }
21             } else { // else part of conditional
22                 do { // make sure move is legal
23                     move = 1+ rand() % 3; // random computer move
24                     } while((move < 1) || (move > 3) || (move > num_objects));
25                     cout << "Computer removed " << move << endl;
26                 }
27                 num_objects = num_objects - move; // implement the move
28                 cout << num_objects << " objects remaining.\n";
29                 current_player = (current_player + 1) % 2; // switch players
30             } while (num_objects > 0);
31         // ----- end of the game loop -----
32         cout << "Player " << current_player << " wins!!!\n";
33         cin.ignore();
34         cout << "\nPress enter to quit.\n";
35         cin.ignore();
36         return 0;
37     }

```

Message 9, control structure body

emac@shams

File Edit Options Buffers Tools Complete In/Out Signals Help

Save Undo

\$ **prog9.cpp**

Compilation of: (prog9.cpp)

Error message: look at line 38, in the program "prog9.cpp".
Missing close parenthesis ')' of for header before the word/charater: "cout".

\$ |

Assume that the actual errors are that the user did not write the close parentheses of the for header, write the expression (i+) instead of (i++), and variable i was not declared in line 37.

```

1
2     #include<iostream>
3     #include<cstdlib>
4     #include<ctime>
5     using namespace std;
6
7     // get_num() asks the user for a
8     // positive number and returns it
9     int get_num();
10    // roll() takes an integer as an
11    // argument, rolls that many dice
12    // randomly and returns the sum
13    int roll(int);
14
15    int main(){
16        int num_dice,sum;
17        srand(time(0));
18        // get number of dice from user
19        num_dice = get_num();
20        // roll dice
21        sum = roll(num_dice);
22        // print sum
23        cout << "Rolled a: " << sum << endl;
24        return 0;
25    }
26
27    int get_num(){
28        int choice = 0;
29        cout << "please enter a positive value: ";
30        cin >> choice;
31        return choice;
32    }
33
34    int roll(int n){
35        int die = 0;
36        int i;
37        for( int i = 0; i < n; i+
38            cout << i << " " << die << " ";
39            die = die + (rand() % 6)+1;
40
41            cout << endl;
42            return die;
43    }

```

5.8 Conclusion

This chapter explained how the 3-phase parsing works. First, it discussed the limitation of LR parsers in detecting errors and justified the need for a solution such as the developed new solution. Then, it showed the code for Phase One and Phase Two. Presenting the parsers' code will help other researchers and developers to advance the new solution. Finally, the sample error messages in section 5.4 showed the results of running this code and solution. Next, empirical experiments with actual novice coders are used to measure whether the error messages generated are actually beneficial.

Chapter 6: Evaluation of Error Message Quality Enabled by 3-Phase Parsing Techniques

6.1 Introduction

This study evaluated the innovative model of 3-phase parsing techniques with an experimental approach. The design of the experiment followed the guidelines from Jonathan Lazar et al.'s book "Research Methods in Human-Computer Interaction" [49] and Creswell and Creswell's book "Research Design: Qualitative, Quantitative and Mixed Methods Approaches" [50].

6.2 Methodology

The experiment aims to evaluate the quality of syntax error messages of the 3-phase parsing techniques. It used EduCC an Educationally Customized Compiler, to generate the error messages. EduCC is a compiler prototype for the C++ language, we developed for this study that implements the 3-phase parsing techniques. The experiment compared the error messages of EduCC with the error messages of the mainstream compilers used in the introductory programming course: GNU GCC version 10.3.1 (GCC) and Microsoft Visual C++ Compiler version 2019 (MSVC).

The study is a controlled experiment and within-group design. The independent variable is the compiler type (EduCC, GCC, MSVC). The dependent variable is the quality of syntax error messages. The quality of syntax error messages is measured by three factors: the success rate of finding errors in erroneous programs, the success rate of fixing syntax errors in erroneous programs and mean-time-to-find and -fix erroneous programs.

The within-group design requires each participant to be exposed to multiple experimental conditions. So, each participant was required to find and fix errors in a program using accompanying error messages from MSCV or GCC as the control group. Then, find and fix errors in another program using error messages from EduCC as the intervention group. The order of programs and compilers was randomized using Qualtrics, as shown in Table 6-3. The participants were assigned randomly to a different set of programs.

6.2.1 Theses

The Null hypotheses for the experiment are:

Hypothesis 1: There is no significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in *finding* syntax errors.

Hypothesis 2: There is no significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in *fixing* syntax errors.

Hypothesis 3: There is no significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in the *time-to-find and -fix*.

The alternative hypotheses for the experiment are:

Hypothesis 1: There is a significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in *finding* syntax errors.

Hypothesis 2: There is a significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in *fixing* syntax errors.

Hypothesis 3: There is a significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in the *time-to-find and -fix*.

6.2.2 Participant

The participants were undergraduate and graduate students enrolled in lower- and upper-division computer science courses at the New Mexico Institute of Mining and Technology in Fall of 2022. Invitations were sent to all the students in the Computer Science department. 66 participants responded to the invitation. After cleaning collected data from empty records, the number of respondents became 53. Table 6.1 shows the age and gender of the participants. Table 6.2 shows the participant's experience in programming.

Table 6.1 Age and gender of the participants.

age	Gender	Count
18 - 24	Man	35
18 - 24	Non-binary / third gender	2
18 - 24	Prefer not to say	2
18 - 24	Woman	7
25 - 34	Man	4
35 - 44	Woman	1
45 - 54	Man	2

Table 6.2 The participants programming experiences.

person-months of coursework or professional experience	C/C++	Python	Java	Others
	count of participants	count of participants	count of participants	count of participants
1-6 months	24	16	14	12
7-12 months	10	12	11	7
13-24 months	8	5	6	1
25-36 months	10	1	-	1
> 36 months	1	-	2	-

6.2.3 Instrument

Part one: experiment tasks: find and fix erroneous programs in C++ using accompanying error messages.

The experiment tasks required the participants to find and fix errors in a set of erroneous programs written in the C++ programming language. Each program is provided with error messages from one of the compilers (GCC, MSVC, or EduCC). The time that the participant used to find and fix errors was recorded. There was no bound on how much time the participant had to find and fix errors. After that, there was an optional second part, where the participants were asked to reflect on their experience with the presented error messages for finding and fixing the erroneous programs.

The experiment used nine programs from Soule's textbook [51]. The nine programs were seeded with common syntax errors. The programs used are equivalent in complexity. Moreover, Program 1, Program 2, and Program 3 were seeded with the same error, a syntax error in the function boundaries. Program 4, Program 5, and Program 6 were seeded with errors in the syntax of control structure headers. Program 7, Program 8, and Program 9 were seeded with errors in the syntax of control structure bodies (see Appendix A for the list of programs). Also, the same program was used once with GCC or MSVC error messages and another with the EduCC error messages.

Since the experiment design is within-group, each participant finds and fixes an erroneous program with GCC or MSVC error messages and another with EduCC error messages. Moreover, some participants start with a program with GCC/MSVC error messages, and others start with EduCC error messages. Table 6.1 shows the groups, programs, compiler, error type, and the number of respondents.

Qualtrics was used for the experiment implementation. It included two parts; part one had four pages. The first page presented the consent form (see section A.2 of Appendix A). The second page

presented three questions about age group, gender, and programming experience (see section A.3 of Appendix A). The third page presented a program with accompanying error messages. The name of the compiler was hidden in the presented messages. Then, the page presented the following questions:

Q1: What is the error in the progX.cpp?

Q2: In which line is the error?

Q3: What is the cause of the error?

Q4: How to fix the error?

The fourth page is like the third page, with different programs and accompanying error messages generated by a different compiler. Pages three and four each have a timer to record the time to find and fix the program. Figure 6.1 shows an example of pages three and four. Furthermore, the workflow features of Qualtrics were used to design the groups and assign respondents randomly to the groups.

Part two: experiment tasks: reflect on the participant's experience with error messages after finishing part one.

After the participant finished the first part of the experiment, they were asked if they wanted to continue to the second part. Part two asked the participants to reflect on their experience with error messages in the first part of the experiment. Part two has four questions. Questions five and six are open ended, and question four asked how friendly the error messages were. However, only questions one, two, and three are covered in this chapter. These questions asked the participants whether the error messages helped them find and fix the errors. It presented the same programs and the accompanying error messages they worked on in part one but showed the actual errors as in Figure 6.2. Then it asks the following yes/no questions:

Q1: Do the compiler error messages correctly give the location (line) of the actual error?

Q2: Do the compiler error messages describe what is the actual error correctly?

Q3: Do the compiler error messages suggest how to fix the error?

6.2.4 Pilot Study

This experiment was designed based on our experience with a pilot experiment. The participants in the pilot experiment were six undergraduate students in lower-division computer science courses at the University of Idaho in the Spring of 2022. The experiment design required each participant to find

and fix errors in all the nine programs instead of just two of the programs. Moreover, in the second part, they reflect on their experiences with error messages. We designed it on Canvas Learning Management System. The expected time for the participant to answer the questions was 90 minutes.

The pilot study faced many obstacles that influenced the version of the experiment design that was ran at the New Mexico Institute of Mining and Technology. For example, the experiment's long time significantly hindered recruiting sufficient participants. Furthermore, the respondents who answered the experiment got tired or bored; some participants skipped half of the questions, and some wrote the same answers to a group of questions. As a result, we redesigned the experiment so that the expected time was 20-30 minutes. This reduction in time was primarily achieved by having the participants answer questions about two programs instead of nine.

6.2.5 Procedure

1. The experiment was designed as an online self-administered questionnaire using Qualtrics.
2. The researcher got the institutional review boards (IRB) exemption from the New Mexico Institute of Mining and Technology (NMT) for the human-subject experiment.
3. Dr. Jeffery, the chair of the Computer Science department, sent the letter of invitation to his department's students via email. The invitation includes a web link to the Qualtrics online questionnaire. Also, three faculty encourage their students in four courses to answer the questionnaire and sent the invitation using Canvas.
4. The questionnaire asked demographic questions, including years of programming experience and proficiency in programming languages.
5. Data was collected anonymously.
6. Participants generally needed 25-30 minutes to complete our study, but no time constraint was forced.

6.2.6 Data Collection

7. The 66 participants' answers were downloaded from the Qualtrics website as a CSV file. Then, we designed a database system in MS Access for the study and imported the data from the Qualtrics CSV file. Some of the rows were empty, so they were deleted. As a result, the records were reduced to 53.
8. Each group's answers were moved to a table in the database.
9. The answers were graded by the researcher.
10. Descriptive and inferential analysis were done using SQL queries of MS Access.
11. Reports of accumulative data was formatted using MS Excel.

Table 6.3 The groups, programs, compiler, type of error, and number of respondents.

Group		Program	Compiler	Type of error: syntax error in the	Number of respondents
group1	1st program	prog1.cpp	GCC	function body	5
	2 nd program	prog3.cpp	EduCC		
group2	1st program	prog2.cpp	MSVC	function body	5
	2 nd program	prog3.cpp	EduCC		
group3	1st program	prog4.cpp	GCC	control structure header	6
	2 nd program	prog6.cpp	EduCC		
group4	1st program	prog5.cpp	MSVC	control structure header	4
	2 nd program	prog6.cpp	EduCC		
group5	1st program	prog7.cpp	GCC	control structure body	6
	2 nd program	prog9.cpp	EduCC		
group6	1st program	prog8.cpp	MSVC	control structure body	5
	2 nd program	prog9.cpp	EduCC		
group7	1st program	prog1.cpp	EduCC	function body	4
	2 nd program	prog2.cpp	GCC		
group8	1st program	prog1.cpp	EduCC	function body	4
	2 nd program	prog3.cpp	MSVC		
group9	1st program	prog4.cpp	EduCC	control structure header	5
	2 nd program	prog5.cpp	GCC		
group10	1st program	prog4.cpp	EduCC	control structure header	5
	2 nd program	prog6.cpp	MSVC		
group11	1st program	prog7.cpp	EduCC	control structure body	6
	2 nd program	prog8.cpp	GCC		
group12	1st program	prog7.cpp	EduCC	control structure body	4
	2 nd program	prog9.cpp	MSVC		

prog7_edu_findfix

Consider the following C++ program (prog7.cpp) and accompanying compiler error messages to answer the following questions:

```

1  #include<iostream>
2  #include<cstdlib>
3  using namespace std;
4
5  int main(){
6      // variables
7      int computerguess;
8      int HorL=1;
9      // instructions to user
10     {cout << "Pick a number (1-100) for the computer to guess.\n";
11     cout << "Don't forget the number.\n";
12     // 1) computer's first guess
13     computerguess = 1 + rand() % 100;
14     while(HorL !=3)
15     }
16     cout << "I guessed: " << computerguess << endl;
17     cout << "(1)higher or (2)lower or (3)correct?\n";
18     cin >> HorL;
19     // user says higher/lower or correct
20     if(HorL == 1){
21         computerguess = computerguess + 1;
22     }
23     if(HorL == 2){
24         computerguess = computerguess - 1;
25     }
26     if(HorL == 3){
27         cout << "Yay, I was right. Computers rock!\n";
28     }
29     // computer guesses again
30     // loop to 1 if not correct
31     // message
32 }

```

When we compiled the previous code (prog7.cpp), we got the following error messages:

What is the error in the prog7.cpp?

In which line is the error?

What is the cause of the error?

How to fix the error?

Figure 6.1 Example of page three/four of the Qualtrics web page for the experiment.

p2_prog5_MCV_q1

Consider prog5.cpp and the accompanying error messages that you saw earlier.

Assume that the actual error in the code is that the user missed the semicolons of the for statement header in line 35, and the variable "i" was not declared.

```

1  #include<iostream>
2  #include<cstdlib>
3  #include<ctime>
4  using namespace std;
5
6  // get_num() asks the user for a
7  // positive number and returns it
8  int get_num();
9  // roll() takes an integer as an
10 // argument, rolls that many dice
11 // randomly and returns the sum
12 int roll(int);
13
14 int main(){
15     int num_dice,sum;
16     srand(time(0));
17     // get number of dice from user
18     num_dice = get_num();
19     // roll dice
20     sum = roll(num_dice);
21     // print sum
22     cout << "Rolled a: " << sum << endl;
23     return 0;
24 }
25
26 int get_num(){
27     int choice = 0;
28     cout << "please enter a positive value: ";
29     cin >> choice;
30     return choice;
31 }
32
33 int roll(int n){
34     int die = 0;
35     for(i < n){ // assume it should be for (int i=1 ; i<n ;
36         die = die + (rand() % 6)+1;
37         cout << i << " : " << die << " ";
38     }
39     cout << endl;
40     return die;
41 }

```

When we compile the previous code (prog5.cpp) using a well-known C++ compiler, we got the following compiler error messages:

Code	Description	Project	File	Line	Suppression State
E0864	i is not a template	ErrExample	prog5.cpp	35	
E0028	expression must have a constant value	ErrExample	prog5.cpp	35	
E0439	expected a '>'	ErrExample	prog5.cpp	35	
C2065	i: undeclared identifier	ErrExample	prog5.cpp	35	
C2143	syntax error: missing ';' before ')'	ErrExample	prog5.cpp	35	
C2143	syntax error: missing ';' before '{'	ErrExample	prog5.cpp	35	
C2143	syntax error: missing ')' before '{'	ErrExample	prog5.cpp	35	
C2065	i: undeclared identifier	ErrExample	prog5.cpp	37	

Do the compiler error messages correctly give the location (line) of the actual error?

Yes, they do

No, they do not

Figure 6.2 Example of part two of the Qualtrics web page for the experiment.

6.3 Results

To evaluate the "helping in finding errors" quality of error message, three questions were used:

- 1) what is the error in the program?
- 2) in which line is the error?
- 3) what is the cause of the error?

To evaluate the "helping in fixing errors" quality, one question was used "how to fix the error?" Also, Qualtrics's timer for each program was employed, which calculates the time the participant spent on the page until the last click.

The success rate for each question are presented in Tables 6.3-6.8 and Figures 6.6-6.9. For the inference analysis, the paired-sample t-test with one-tail was used. The grades of questions were used to calculate the difference between the control and intervention groups' answers. Correct answer was graded as 2. Partially correct answer was graded as 1. Incorrect answer was graded as 0. There was one record that the participant left one question blank and answered the other three questions for a program; it was graded as 0 because the assumption was that they failed to answer it.

RQ1: Is there a significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in *finding* syntax errors?

Table 6.4 shows the success rate of answering question one, "what is the error in the program?". As can be seen, the average success rate for EduCC is higher than GCC and MSVC. Also, the poorest performance for GCC and MSVC was answering question three, "What is the cause of the error?" as shown in Table 6.6. On the other hand, the average success rate for the three compilers is acceptable for answering question two, "In which line is the error?" as in Table 6.5. Moreover, these results agree with the participants' answers in part two when they reflect on their experience with error messages in part one. Figure 6.3 shows a chart summarizing participants' answers to question one, "Do the compiler error messages correctly give the location (line) of the actual error?". Figure 6.4 shows a chart summarizing the participants' answers to question two, "Do the compiler error messages describe the actual error?"

Table 6.4 Success rate of answering the question “what is the error in the program?”

Question 1: What is error in the program?									
Compiler	EduCC	EduCC	EduCC	GCC	GCC	GCC	MSVC	MSVC	MSVC
Category of Answers	Correct	Incorrect	Partial Correct Answer	Correct	Incorrect	Partial Correct Answer	Correct	Incorrect	Partial Correct Answer
Count of participant who answer according to the category	44	2	7	14	11	6	12	6	4
Success rate of answering find question 1 (%)	83%	4%	13%	45%	35%	19%	55%	27%	18%

Table 6.5 Success rate of answering the question “in which line is the error?”

Question 2: In which line is the error?									
Compiler	EduCC	EduCC	EduCC	GCC	GCC	MSVC	MSVC	MSVC	MSVC
Category of Answers	Correct	Incorrect	Partial Correct Answer	Correct	Incorrect	Correct	Incorrect	Partial Correct Answer	Partial Correct Answer
Count of participant who answer according to the category	42	9	2	24	7	16	4	2	2
Success rate of answering find question 1 (%)	79%	17%	4%	77%	23%	73%	18%	9%	9%

Table 6.6 Success rate of answering the question “what is the cause of the error?”

Question 3: What is the cause of error?										
Compiler	EduCC	EduCC	EduCC	GCC	GCC	GCC	GCC	MSVC	MSVC	MSVC
Category of Answers	Correct	Incorrect	Partial Correct Answer	Correct	Incorrect	Left Blank	Partial Correct Answer	Correct	Incorrect	Partial Correct Answer
Count of participant who answer according to the category	49	2	2	18	9	1	3	16	5	1
Success rate of answering find question 1 (%)	92%	4%	4%	58%	29%	3%	10%	73%	23%	5%

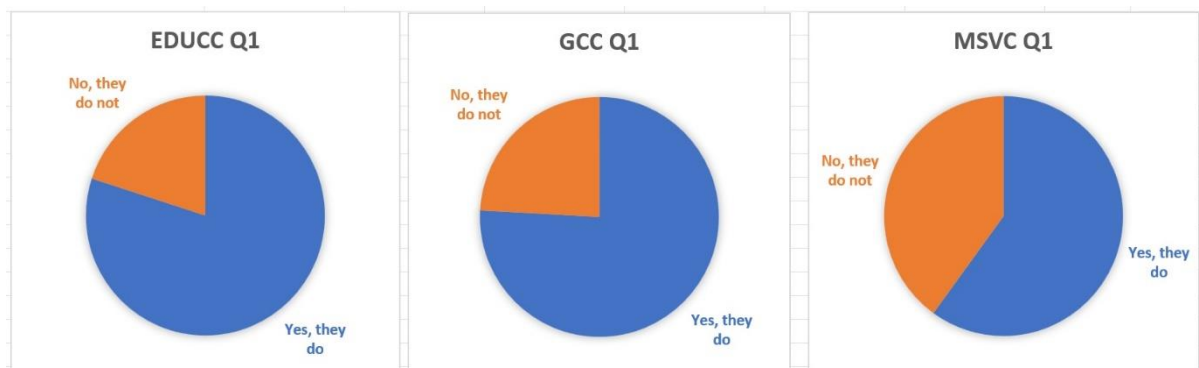


Figure 6.3 Participants’ answers for the question "Do the compiler error messages correctly give the location (line) of the actual error?"

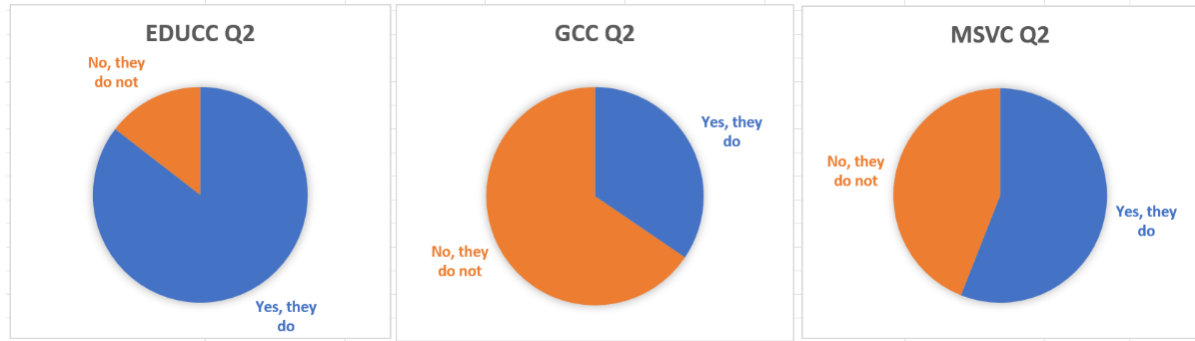


Figure 6.4 Participants' answers for the question " Do the compiler error messages describe what is the actual error?"

Finally, a paired-sample t-test suggests that there is a significant difference between the mean of participants' answers when they used accompanying error messages from EduCC and when they used accompanying error messages from GCC or MSVC for finding errors ($t= 4.548825$, $df=52$, $p = 1.63228E-05$).

RQ2: Is there a significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in *fixing* syntax errors?

Table 6.7 shows the success rate of answering "How to fix the error?". As can be seen, the average success rate for EduCC is higher than GCC and MSVC. Participants' performance answering "how to fix the error" is acceptable. The average rate is higher than 50%. However, the reflection of participants in part two clarifies that they do not think that the accompanying error messages from GCC and MSVC were helpful, as shown in Figure 6.5.

Table 6.7 Success rate of answering the question "how to fix the error?"

Question 4: How to fix the error?											
Compiler	EduCC	EduCC	EduCC	EduCC	GCC	GCC	GCC	GCC	MSVC	MSVC	MSVC
Category of Answers	Correct	Incorrect	Left Blank	Partial Correct Answer	Correct	Incorrect	Left Blank	Partial Correct Answer	Correct	Incorrect	Partial Correct Answer
Count of participant who answer according to the category	47	3	1	2	17	9	1	4	16	5	1
Success rate of answering find question 1 (%)	89%	6%	2%	4%	55%	29%	3%	13%	73%	23%	5%

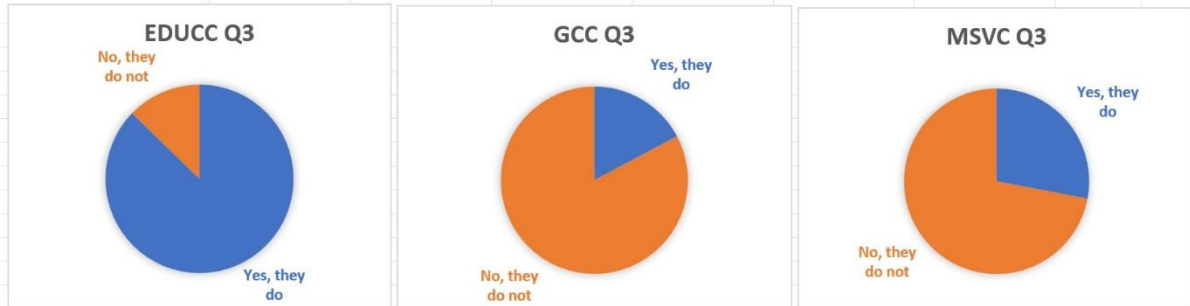


Figure 6.5 Participants' answers for the question "Do the compiler error messages suggest how to fix the error?"

A paired-sample t-test suggests that there is a significant difference between the mean of participants' answers for the question "how to fix the error?" when they used accompanying error messages from EduCC and when they used accompanying error messages from GCC or MSVC ($t = 3.954749$, $df = 52$, $p = 1.16408 \text{ E-}04$).

RQ3: Is there a significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in the *time-to-find and -fix*?

A paired-sample t-test suggests that there is no significant difference in the task completion time between the time used to find and fix error when the participants used accompanying error messages of EduCC, and the time used when they used accompanying error messages of GCC or MSVC. ($t = -1.63144$ while the critical t-value is -1.68 , $df = 52$, $p = 0.054419$).

6.4 Limitations

This study compared EduCC with versions of MSVC (2019) and GCC (10.3.1), which may have new versions and enhanced error messages in the future. It was very difficult to recruit students to spend hours on a lengthy experiment. This happened in the pilot study, so we were forced to shorten the experiment time for the full experiment.

In implementing the experiment, it was difficult to separate the time to find errors from the time to fix the error using Qualtrics. So, developing a specialized environment for studying programmers' performance will be more helpful in the future. Finally, this study shows how to improve some common syntax errors for novices, but the more advanced students may need help with semantics and run-time errors.

6.5 Conclusion

The research question of this dissertation is:

Can modified parsing techniques help in generating better syntax error messages?

This question was evaluated using a controlled experiment and within-group design. The independent variable is the compiler type (EduCC, GCC, MSVC). The dependent variable is the quality of syntax error messages. The quality of syntax error messages is measured by three factors: the success rate of finding errors in erroneous programs, the success rate of fixing syntax errors in erroneous programs, and mean-time-to-find and -fix erroneous programs.

The Null hypotheses for the experiment are:

Hypothesis 1: There is no significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in finding syntax errors.

Hypothesis 2: There is no significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in fixing syntax errors.

Hypothesis 3: There is no significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in the time-to-find and -fix.

The success rate of questions “what is the error?”, “in which line is the error?” and “ what is the cause of the error?” are 83%, 79%, and 92% consequentially for the programs that the participants used EduCC. While 45%, 77%, and 58% for the programs where the participants used GCC, and 55%, 73%, and 73% for the programs that used MSVC. This supports that EduCC enhanced compiler error messages in finding the error. Also, the sample-paired t-test defended Null Hypothesis 1.

The success rate of the question “how to fix the error” is 89% for EduCC, 55%, and 73% for GCC and MSVC, consequentially. This supports that EduCC enhanced compiler error messages in fixing the error. Also, the sample-paired t-test defended Null Hypothesis 2. However, the study did not prove that the EduCC reduced the time to find and fix the error.

Finally, this chapter evaluated the 3-phase parsing techniques using an experimental approach. The results proved that the 3-phase parsing techniques could significantly enhance compiler error messages.

Chapter 7: Conclusion and Future Work

The research question of this dissertation is:

Can modified parsing techniques help in generating better syntax error messages?

This question was evaluated using a controlled experiment and within-group design. The experiments proved that the 3-phase parsing techniques enhanced compiler error messages. As a result, they help the user in finding and fixing errors. Previous studies showed that enhanced compiler error messages impact student learning in introductory programming courses and reduce the number of repeated errors.

This study complemented the efforts of researchers in this area. Many scientists focus on enhancing the structure and wording of the error message contents. Others focus on designing solutions to implement these recommendations. For example, Becker worked on proving that enhanced error messages have a positive impact on students learning. Barik focuses on the structure of the content of the error messages. Marcus suggested a rubric to measure the effectiveness of modified error messages. Kohn studied the connection between misconceptions and error messages and designed a parser to address these misconceptions within the error messages. The contributions of this study are:

1. Analyzing and studying the behavior of mainstream compilers with novices' common errors.
2. Designing an innovative 3-phase parsing technique
3. Developing a proof-of-concept compiler prototype that demonstrates the utility of the 3-phase parsing techniques for C/C++ languages.
4. Developing metrics to measure the quality of syntax error messages.
5. Conducting an experiment to measure the quality of syntax error messages.

The innovative model of 3-phase parsing techniques made it possible to implement some of the recommendations of previous studies. Barik, in his recommendations, indicated that they proved that the rational model of error messages is preferable by experts and that the next research should interview the compiler writers and understand why they do not implement these recommendations about the error messages' content and structure. The 3-phase parser generated in this dissertation can be an answer to Barik and others' recommendations. Furthermore the 3-phase parser utilizes the Merr tool. Merr enables writing good error message content that is connected to the parsing states. Using multiple parsing techniques gives the compiler writers more control over writing more specific error messages for each parsing state and closer to the location of the errors.

Kohn developed a parser to report error messages for 80% of predefined common errors. However, Kohn reported that the major limitation of their approach is the need to hard code the predefined errors that connected to students' misconceptions. They cannot always write the correct message for the error since they cannot guess what the students thought when they wrote the code. And for the same error, different students may need different explanations. The 3-phase parser is a more general approach that depends on the parsing states, not writing specific code for each error. Also, Kohn's parser works only on small programs for a subset of Python grammars. Whereas the 3-phase parser should not be limited by the program size. Future studies should test the 3-phase parser on large programs. Also, it is expected that the 3-phase parsing techniques have the potential to be applied to the other programming languages, especially those that are from the C language family, due to the similarities in their grammars.

Future research

For future work, I plan to work on the other subsystems of the proposed ILDE: software visualization tool, multimedia learning content production, eTutoring tool, and ILDE's mascot character. Next, I will integrate these subsystems into the ILDE and test these subsystems with mixed-method studies. Also, I plan to work on two studies; their need arises while working on this research. The first study is developing a goodness rating system for the quality of syntax error messages. The second study is developing a practical syntax error message coverage tool.

Undergraduate Students Co-design Educational Compiler Error Messages

The research project aims to provide insight into how compiler error messages should be written from undergraduate computer science students' perspectives. The core idea of co-design is co-creation between users and design experts to learn from the collective creativity of potential users. The project aims to involve users of the EduCC Compiler, students, in the innovative design of compiler error messages. Letting students write error messages by themselves can generate more friendly content of error messages. That uses the students' terminology instead of compiler writer jargon. It will give more insight into what the students find helpful for them. The suggested research method is the focus group.

Toward a Goodness Rating System for The Quality of Syntax Error Messages

Not all compiler error messages are equal! This research describes error messages as bad and good quality. Not all good error messages are at the same level. Criteria to measure the quality of error messages include readability, relevancy, compiler writer jargon, consistency, or hints to fix.

Developing metrics for measuring compiler error messages' quality helps compiler writers and researchers. First, it will help compare different compilers' error messages. Then it will enable the researcher to search for causes of low-quality error messages and address these causes. Alternatively, if there is a compiler with better error messages, questioning the source of its strength and sharing it will help other compiler writers craft good error messages.

A plan for this future study may include the following:

1. Investigating the literature for evidence-based research on the quality of compiler error messages.
2. Conducting focus group meetings with compiler writers and teachers of introductory programming courses.
3. Developing a rubric and testing it.
4. Applying this rubric to a set of error messages of different compilers, report and discuss the results.

A Practical Syntax Error Message Coverage Tool

Good compiler error messages are a challenge for LR parsers. Studying the common errors of novices and seeing how the compiler responds to these errors is a good step. However, also needed is the ability to generate erroneous code fragments that lead to each state in the LR parser automaton where error can happen, which is reachability. Offering a practical algorithm and tool that generates a complete set of erroneous snippets for a programming language helps researchers study, enhance, and evaluate how compilers diagnose these erroneous snippets. It also enables compiler writers to craft comprehensive diagnostic syntax error messages.

A plan for this future study may include the following:

1. Designing an algorithm that generates a complete set of erroneous fragments for a language grammar.
2. Developing an open-source tool that implements this algorithm.
3. Testing and verifying generated snippets of this tool on a test suite of LR grammars.

References

- [1] L. Westfall. *Certified Software Quality Engineer Handbook*. ASQ Quality Press, pp.4-6, 2009, [E-book] Available: Ebook Central Academic Complete.
- [2] S. M. Algaraibeh, “Techniques for enhancing compiler error messages,” in Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 2, 2022, pp. 1–2.
- [3] R. Molich, and J. Nielsen, “Improving a Human-Computer Dialogue: What Designers Know about Traditional Interface Design”. vol. 33, no. 3, ACM, New York, 1990, pp. 338-348.
- [4] S. Hyunmin, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, “Programmers’ build errors: A case study (at Google)”. In International Conference on Software Engineering (ICSE). 2014, pp. 724–734.
- [5] B. Becker, P. Denny, R. Pettit, D. Bouchard, D. Bouvier, B. Harrington, A. Kamil, A. Karkare, C. McDonald, P. Osera, J. Pearce, and J. Prather. “Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research”. In the Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education, ITiCSE-WGR '19. ACM, New York, 2019, pp. 177–210.
- [6] B. Shneiderman. “Designing Computer System Messages”. ACM, New York, Sep 1982, pp. 610–611.
- [7] P. Brown. “Error messages: the neglected area of the man/machine interface”. ACM, New York, 1983, pp. 246–249.
- [8] S. Lewis and G. Mulley.” A Comparison Between Novice and Experienced Compiler Users in a Learning Environment”. In Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual Conference on Integrating Technology into Computer Science Education: Changing the Delivery of Computer Science Education (ITiCSE '98). ACM, New York, 1998, pp.157–161.
- [9] V. Traver, "On Compiler Error Messages: What They Say and What They Mean." Advances in Human-computer Interaction 2010, Hindawi Publishing Corporation, June 2010, Available: doi:10.1155/2010/602570
- [10] G. Marceau, K. Fisler, and S. Krishnamurthi. “Measuring the Effectiveness of Error Messages Designed for Novice Programmers”. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11). ACM, New York, 2011, pp. 499-504.
- [11] T. Barik, D. Ford, E. Murphy-Hill, and C. Parnin, “How Should Compilers Explain Problems to Developers?”, In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). ACM, New York, 2018, pp.633–643.
- [12] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin, “Do developers read compiler error messages?”, In International Conference on Software Engineering (ICSE), 2017, 575–585.
- [13] B. Becker, P. Denny, J. Prather, R. Pettit, R. Nix, and C. Mooney, “Towards Assessing the Readability of Programming Error Messages”, 2021, ACE '21 The 23rd Australasian Computing Education Conference, 2021.
- [14] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, “Identifying and Correcting Java Programming Errors for Introductory Computer Science Students,” ACM Special Interest Group Computer Science Education (SIGCE) Bulletin, ACM, vol. 35, no. 1, New York, 2003, pp. 153–156.
- [15] T. Kohn, *Teaching python programming to novices: addressing misconceptions and creating a development environment*. ETH Zurich, 2017.

- [16] C. Jeffery, "Generating LR syntax error messages from examples", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, no. 5, ACM, New York, 2003, pp. 631-640.
- [17] F. Pottier, "Reachability and error diagnosis in LR(1) parsers", In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, New York, 2016, pp. 88-98.
- [18] B. A. Becker, G. Glanville, R. Iwashima, C. McDonnell, K. Goslin, and C. Mooney, "Effective compiler error message enhancement for novice programming students," *Computer Science Education*, vol. 26, no. 2-3, 2016, pp. 148-75.
- [19] T. Barik, *Error messages as rational reconstructions*. North Carolina State University, 2018.
- [20] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, "A multi-national, multi-institutional study of assessment of programming skills of first-year cs students," in *Working group reports from ITiCSE on Innovation and technology in computer science education*, ACM, New York, 2001, pp. 125-180.
- [21] J. Sweller, J. van Merriënboer and F. Paas, "Cognitive architecture and instructional design: 20 years later", *Educ Psychol Rev*, vol. 31, pp. 261-292, Jan. 2019.
- [22] A. J. Martin and P. Evans, "Load reduction instruction: Sequencing explicit instruction and guided discovery to enhance students' motivation engagement learning and achievement," in *Advances in Cognitive Load Theory*. London and New York, UK and US:Routledge, pp. 13-29, 2020.
- [23] D. A. Kolb, "The process of experiential learning," in *Experiential Learning: Experience as the Source of Learning and Development*. FT Press, 2014.
- [24] C. Bonk and D. Cunningham, "Searching for learner-centered constructivist and sociocultural components of collaborative educational learning tools," in *Electronic Collaborators: Learner-Centered Technologies for Literacy apprenticeship, and discourse*. Routledge, New York, 1998, pp. 25-50.
- [25] R. E. Mayer, "Cognitive theory of multimedia learning", in *The Cambridge handbook of multimedia learning*. Cambridge university press, 2014, pp. 43-71.
- [26] U. Fuller et al., "Developing a computer science-specific learning taxonomy", *ACM Special Interest Group Computer Science Education (SIGCE) Bulletin*, vol. 39, no. 4, ACM, New York, 2007, pp. 152-170.
- [27] B. Du Boulay, "Some difficulties of learning to program", *Journal of Educational Computing Research*, vol. 2, no. 1, SAGE Publications, Los Angeles, CA, 1986, pp. 57-73.
- [28] A. F. Blackwell, "First steps in programming: A rationale for attention investment models", *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, 2002, pp. 2-10.
- [29] J. Bonar and E. M. Soloway, "Pre-programming knowledge: A major source of misconceptions in novice programmers", *Human-Computer Interaction*, vol. 1, Taylor & Francis, 1985, pp. 133-162.
- [30] A. Ettles, A. Luxton-Reilly and P. Denny, "Common logic errors made by novice programmers", *Proceedings of the 20th Australasian Computing Education Conference (ACE)*, ACM, New York, 2018, pp. 83-89.
- [31] L. Kaczmarczyk, E. Petrick, J. East and G. Herman, "Identifying student misconceptions of programming", *Proceedings of the 41st ACM technical symposium on Computer science education*, ACM, New York, pp. 107-111.
- [32] T. Green, "Language design and acquisition of programming: Programming languages as information structures" in *Psychology of Programming*, Academic Press, 1990 pp. 117-137.

- [33] J. Hoc and A. Nguyen-Xuan, "Language design and acquisition of programming: Language semantics mental models and analogy" in *Psychology of Programming*, Academic Press, 1990, pp. 139-156.
- [34] A. Altadmri and N. Brown, "37 million compilations: Investigating novice programming mistakes in large-scale student data", *Proceedings of the 46th ACM technical symposium on computer science education*, pp. 522-527, 2015.
- [35] T. Kohn, "The error behind the message: Finding the cause of error messages in python", *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019, pp. 524-530.
- [36] M. Nienaltowski, M. Pedroni and B. Meyer, "Compiler error messages: What can help novices?", *Proceedings of the ACM Technical Symposium on Computer Science Education*, 2008, pp. 168-172.
- [37] M. Ahmadzadeh, D. Elliman and C. Higgins, "An analysis of patterns of debugging among novice computer science students", *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, ACM, New York, 2005, pp. 84-88.
- [38] W. L. Johnson and E. Soloway, "PROUST: Knowledge-based program understanding", *IEEE Transactions on Software Engineering*, vol. SE-11, no. 3, 1985, pp. 267-275.
- [39] M. Kolling, B. Quig, A. Patterson and J. Rosenberg, "The BlueJ system and its pedagogy", *Journal of Computer Science Education Special Issue on Learning and Teaching Object Technology*, vol. 13, no. 4, Taylor & Francis 2003.
- [40] S. Fincher, S. Cooper, M. Kölling and J. Maloney, "Comparing alice greenfoot & scratch", *ACM Transactions on Computing Education (TOCE)*, ACM, New York, 2010, pp. 192-193.
- [41] N. Trachsler, "WebTigerJython - A Browser-based programming IDE for education", Master's Thesis, ETH Zurich, 2018.
- [42] T. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework", *Visual Languages and Computing*, Elsevier, 1996 pp. 131-174.
- [43] I. Goldin, S. Narciss, P. Foltz and M. Bauer, "New directions in formative feedback in interactive learning environments", *International Journal of Artificial Intelligence in Education*, vol. 27, no. 3, Springer, 2017, pp. 385-392.
- [44] C. L. Jeffery, "Parsing" in *Build Your Own Programming Language*. Packt Publishing, 2021
- [45] D. Grune and C. J. H. Jacobs, *Parsing Techniques: A Practical Guide*. New York, NY: Springer New York, 2008, pp. 263–341. Available: <https://doi.org/10.1007/978-0-387-68954-89>
- [46] A. V. Aho, R. Sethi, and J. D. Ullman, "LR Parsing" section of "Syntax Analysis" in *Compilers: principles, techniques, and tools*. Addison-wesley Reading, 2007, vol. 2.
- [47] G. G. P. LICENSE, "Bison 2.7," 2007.
- [48] C. L. Jeffery, "Merr user's guide", Accessed: March 23, 2023. [Online]. Available: <http://unicon.org/merr/>, 2002.
- [49] J. Lazar, J. H. Feng, and H. Hochheiser, "Experimental Design" in *Research methods in human-computer interaction*. Morgan Kaufmann, 2017. [Online]
- [50] J. W. Creswell, and J. D. Creswell, "Quantitative Methods" in *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017.
- [51] T. Soule, *A Project-Based Introduction to C++*. Kendall Hunt, 2014, pp. 5, 36, 54, 71, 92.

Appendices

Appendix A: Study Materials for Experimental Approach to Evaluate Messages Enabled by 3-Phase Parsing Techniques

A.1 Invitation Letter

Invitation to Participate in a Compiler Research Experiment

Dear Computer Science and Engineering Students,

You are invited to participate in the "Techniques for Enhancing Compiler Error Messages" experiment. This research aims to enhance the learning and teaching of introductory programming courses. One of programming learners' main difficulties is handling and debugging code errors. The study will evaluate the quality of a new compiler developed by the researchers. Furthermore, the study will compare the quality of the error messages generated by the newly developed compiler with the used compilers by the students of introductory programming courses, GNU GCC and Microsoft Visual C++.

Dr. Clinton Jeffery and Sana'a Algaraibeh are leading this study. Dr. Clinton Jeffery is a professor and chair, and Sana'a Algaraibeh is an instructor in the Computer Science and Engineering Department at the New Mexico Institute of Mining and Technology.

The survey is light and divided into parts. Each part is only 15 minutes. If you do the first part, you will help this research proceed. If you do two parts, you will help make the study more valuable.

Best regards,
Dr. Clinton Jeffery
Chair, Computer Science and Engineering Department
clinton.jeffery@nmt.edu

A.2 Consent Form



Dear Computer Science and Engineering Students,

You are invited to participate in a research study titled "Techniques for Enhancing Compiler Error Messages." Dr. Clinton Jeffery and Sanaa Algaraibeh are leading this study. Dr. Clinton Jeffery is a professor and chair, and Sanaa Algaraibeh is an instructor at the Computer Science and Engineering Department at the New Mexico Institute of Mining and Technology.

About the study
This research aims to enhance the learning and teaching of introductory programming courses. One of programming learners' main difficulties is handling and debugging code errors. The study will evaluate the quality of a new compiler developed by the researcher. The compiler is software that converts source code into machine language and generates error messages. The newly developed compiler aims to enhance the generated compiler error messages, so they will be more helpful to the students while learning how to program. The study will compare the quality of the error messages generated by the newly developed compiler with the used compilers (GNU GCC and Microsoft Visual C++) by the students of introductory programming courses.

What we will ask you to do
You will be asked to find and fix errors in a group of programs. The study has many parts. In some parts, you will find and fix errors in a group of nine erroneous C++ programs using the accompanying compiler error messages. In the other parts, you will evaluate the compiler error messages. We expect the time to participate is 15 minutes for each part. You can choose to answer one or more parts. If you consent to participate, select the "Yes, I consent" below.

Risks and discomforts
The risks and discomforts associated with this study are minimal, no more than normal learning tasks. Therefore, we anticipate that your participation in this survey presents no greater risk than the everyday use of the Canvas Learning Management System.

Benefits
The benefits of participation include developing critical thinking, debugging, and handling errors skills. Also, enhancing compiler error messages will save time and effort for learners and expert programmers.

Your participation is voluntary.
You must be at least 18 years old to participate in the study, and your participation in this research study is voluntary. If you consent to participate and later become uncomfortable or decide you no longer want to participate for any reason, you may withdraw by not submitting the survey. Your participation or non-participation in this study does not impact your standing or grades in class(es) or current/future relationship with course instructors or the New Mexico Institute of Mining and Technology and College of Engineering.

Privacy/Confidentiality/Data Security
The survey is anonymous and will not collect any identification from participants. Therefore, we will not connect your answers with your identity. The study results will be used for scholarly purposes only and shared with university representatives. If you have questions about this study, please contact Dr. Clinton Jeffery and Sanaa Algaraibeh. The New Mexico Institute of Mining and Technology Institutional Review Board has approved this project. If you have any questions about your rights as a research participant, you can contact the New Mexico Institute of Mining and Technology IRB coordinator at irb@nmt.edu.

Researcher	Researcher
Dr. Clinton Jeffery clinton.jeffery@nmt.edu Office: Cramer 230 Computer Science and Engineering Department New Mexico Institute of Mining and Technology 801 Leroy Place Socorro, NM 87801	Sanaa Algaraibeh sanaa.algaraibeh@nmt.edu Office: Cramer 226 Computer Science and Engineering Department New Mexico Institute of Mining and Technology 801 Leroy Place Socorro, NM 87801

I have read the above information and have received answers to any questions I asked. I consent to take part in the study.

Yes, I consent

No, I do not consent

Figure A- 1 shows the consent form, the first page of the experiment on Qualtrics.

A.3 Tasks

Which age group describe you?

18 - 24

25 - 34

35 - 44

45 - 54

> = 55

How do you describe yourself:

Man

Woman

Non-binary / third gender

Prefer not to say

How knowledgeable are you about the following programming languages?

person-months of coursework or professional experience

C/C++

Python

Java

Others, name them

NEXT

0% 100%

Figure A- 2 shows the second page of part 1 of the experiment for group 1 on Qualtrics. First page is same for all the groups.

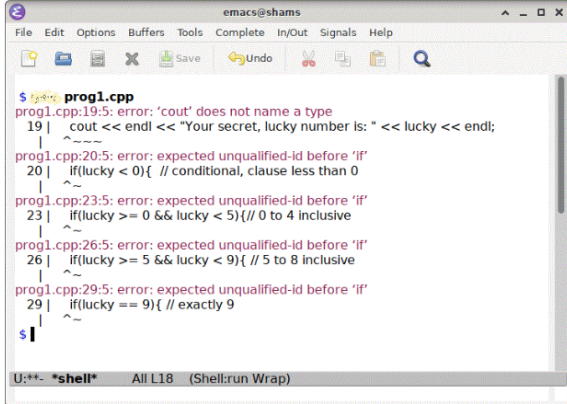
Consider the following C++ program (prog1.cpp) and the accompanying compiler error messages to answer the following questions:

```

1 /* The fortune Teller -
2  * a simple program introducing some
3  * fundamental programming concepts.
4  */
5 #include <iostream> // include a library
6 using namespace std;
7 int main() // main() starts the program
8 {
9     // ----- Variable Declarations -----
10    int favorite; // create a variable to store the favorite number
11    int disliked; // create a variable to store the disliked number
12    int lucky; // create a variable to store the lucky number
13    // ----- Get user input -----
14    cout << "Enter your favorite number (no decimals): "; // output
15    cin >> favorite; // user input
16    cout << "Enter a number you don't like (no decimals): ";
17    cin >> disliked;
18    lucky = ((abs(favorite-disliked))*13) % 10;
19    cout << endl << "Your secret, lucky number is: " << lucky << endl;
20    if(lucky < 0){ // conditional, clause less than 0
21        cout << "Try to be less negative." << endl;
22    }
23    if(lucky >= 0 && lucky < 5){ // 0 to 4 inclusive
24        cout << "Think bigger!" << endl;
25    }
26    if(lucky >= 5 && lucky < 9){ // 5 to 8 inclusive
27        cout << "Today you should embrace technology." << endl;
28    }
29    if(lucky == 9){ // exactly 9
30        cout << "Today is your lucky day!" << endl;
31    }
32    // ----- Code to help the program exit "gracefully" -----
33    cin.ignore();
34    cout << "Press enter to quit." << endl;
35    cin.ignore();
36    return 0;
37 }

```

When we compiled prog1.cpp using a well-known C++ compiler, we got the following error messages:



```

$ g++ prog1.cpp
prog1.cpp:19:5: error: 'cout' does not name a type
19 |     cout << endl << "Your secret, lucky number is: " << lucky << endl;
    |     ^
prog1.cpp:20:5: error: expected unqualified-id before 'if'
20 |     if(lucky < 0){ // conditional, clause less than 0
    |     ^
prog1.cpp:23:5: error: expected unqualified-id before 'if'
23 |     if(lucky >= 0 && lucky < 5){ // 0 to 4 inclusive
    |     ^
prog1.cpp:26:5: error: expected unqualified-id before 'if'
26 |     if(lucky >= 5 && lucky < 9){ // 5 to 8 inclusive
    |     ^
prog1.cpp:29:5: error: expected unqualified-id before 'if'
29 |     if(lucky == 9){ // exactly 9
    |     ^
$

```

U++: *shell* All L18 (Shell:run Wrap)

What is the error in the prog1.cpp?

In which line is the error?

What is the cause of the error?

How to fix the error?

Timing

These page timer metrics will not be displayed to the recipient.

First Click	20.333 seconds
Last Click	69.852 seconds
Page Submit	0 seconds
Click Count	2 clicks

Figure A- 3 shows page 3 of part 1 of the experiment for group 1 on Qualtrics.

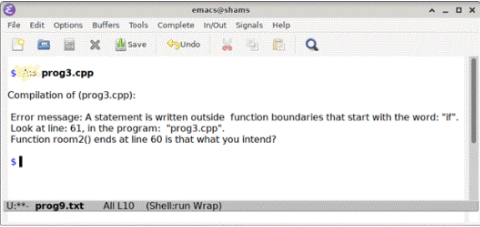
Consider the following C++ program (prog3.cpp) and accompanying compiler error messages to answer the following questions:

```

1 #include<iostream>
2 using namespace std;
3
4 // Function prototypes
5 int room1(); // addition
6 int room2(); // subtraction
7 //int goldstars = 0; // global variable, can be 'seen' in all functions.
8
9 int main(){
10 int current_room = 1; // local variable, only exists in main()
11 cout << "This is a math, adventure game.\n";
12 do{
13     cout << endl << endl;
14     cout << "current_room: " << current_room << endl;
15     cout << "Gold stars: " << goldstars << endl;
16
17     switch(current_room){
18     case 1:
19         current_room = room1();
20         break;
21     case 2:
22         current_room = room2();
23         break;
24     default:
25         cout << "Error: " << current_room << endl;
26     }
27 }while(current_room != 3);
28 cout << "You win, Bye!\n";
29 }
30
31 // function definitions (code)
32 int room1(){
33     static int visited = 0; // keeps track of whether this room has been visited
34     already
35     int choice;
36     if(visited == 0){ // if not visited, get a gold star
37         goldstars = goldstars + 1;
38         visited = 1; // change the value so next time it counts as visited
39     }
40     cout << "This is room 1.\n";
41     cout << "1) stay here\n";
42     cout << "2) go to the subtraction room.\n";
43     cin >> choice;
44     if (choice == 1){
45         return 1; // 'goto' room 1
46     }
47     if (choice == 2){
48         return 2; // goto room 2
49     }
50 }
51 int room2(){
52     int choice;
53     cout << "The subtraction room (2).\n";
54     cout << "1) stay\n";
55     cout << "2) room 1\n";
56     cout << "3) exit\n";
57     cin >> choice;
58     if(choice == 1)
59         return 2; // stay in room 2
60     }
61     if(choice == 2){
62         return 1; // go to room 1
63     }
64     if(choice == 3){
65         return 3; // will exit the game
66     }

```

When we compiled the previous code (prog3.cpp), we got the following error messages:



U+*: prog9.txt All L10 (Shell:run Wrap)

What is the error in the prog3.cpp?

In which line is the error?

What is the cause of the error?

How to fix the error

Figure A- 4 shows page 4 of part 1 of the experiment for group 1 on Qualtrics.



Consider prog1.cpp and the accompanying error messages that you saw earlier.

Assume that the actual error in the code is that the user added a close curly bracket at the end of line 18. As a result, the statements located after this close curly bracket became outside the function boundaries.

```

1  /* The fortune Teller -
2   * a simple program introducing some
3   * fundamental programming concepts.
4   */
5  #include<iostream> // include a library
6  using namespace std;
7  int main() // main() starts the program
8  {
9   // ----- Variable Declarations -----
10  int favorite; // create a variable to store the favorite number
11  int disliked; // create a variable to store the disliked number
12  int lucky; // create a variable to store the lucky number
13  // ----- Get user input -----
14  cout << "Enter your favorite number (no decimals): "; // output
15  cin >> favorite; // user input
16  cout << "Enter a number you don't like (no decimals): ";
17  cin >> disliked;
18  lucky = ((abs(favorite-disliked))*13) % 10;
19  cout << endl << "Your secret, lucky number is: " << lucky << endl;
20  if(lucky < 0){ // conditional, clause less than 0
21  cout << "Try to be less negative." << endl;
22  }
23  if(lucky >= 0 && lucky < 5){ // 0 to 4 inclusive
24  cout << "Think bigger!" << endl;
25  }
26  if(lucky >= 5 && lucky < 9){ // 5 to 8 inclusive
27  cout << "Today you should embrace technology." << endl;
28  }
29  if(lucky == 9){ // exactly 9
30  cout << "Today is your lucky day!" << endl;
31  // ----- Code to help the program exit "gracefully" -----
32  cin.ignore();
33  cout << "Press enter to quit." << endl;
34  cin.ignore();
35  return 0;
36}

```

When we compiled the previous code using a well-known C++ compiler, we got the following error messages:

```

$ emacs@shams
File Edit Options Buffers Tools Complete In/Out Signals Help
$ prog1.cpp
prog1.cpp:19:5: error: 'cout' does not name a type
19 | cout << endl << "Your secret, lucky number is: " << lucky << endl;
   | ^~~~~
prog1.cpp:20:5: error: expected unqualified-id before 'if'
20 | if(lucky < 0){ // conditional, clause less than 0
   | ^~
prog1.cpp:23:5: error: expected unqualified-id before 'if'
23 | if(lucky >= 0 && lucky < 5){ // 0 to 4 inclusive
   | ^~
prog1.cpp:26:5: error: expected unqualified-id before 'if'
26 | if(lucky >= 5 && lucky < 9){ // 5 to 8 inclusive
   | ^~
prog1.cpp:29:5: error: expected unqualified-id before 'if'
29 | if(lucky == 9){ // exactly 9
   | ^~
$ |
U:++- *shell* All L18 (Shell:run Wrap)

```

Figure A- 5 shows page 1 of part 2 of the experiment for group 1 on Qualtrics.

Do the compiler error messages correctly give the location (line) of the actual error?

Yes, they do

No, they do not

Do the compiler error messages describe what is the actual error correctly?

Yes, they do

No, they do not

Do the compiler error messages suggest how to fix the error?

Yes, they do

No, they do not

Figure A- 6 shows the questions of page 1 of part 2 of the experiment for group 1 on Qualtrics.

Consider prog3.cpp and the accompanying error messages that you saw earlier.

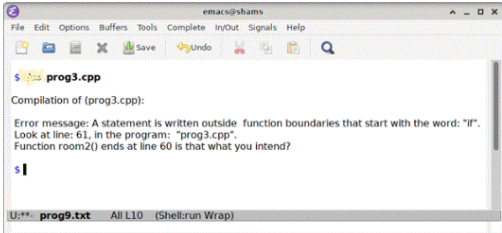
Assuming the actual error in the code is that the user added a close curly bracket in line 60.

```

1  #include<iostream>
2  using namespace std;
3
4  // Function prototypes
5  int room1(); // addition
6  int room2(); // subtraction
7  //int goldstars = 0; // global variable, can be 'seen' in all functions.
8
9  int main(){
10     int current_room = 1; // local variable, only exists in main()
11     cout << "This is a math, adventure game.\n";
12     do{
13         cout << endl << endl;
14         cout << "current_room: " << current_room << endl;
15         cout << "Gold stars: " << goldstars << endl;
16
17         switch(current_room){
18             case 1:
19                 current_room = room1();
20                 break;
21             case 2:
22                 current_room = room2();
23                 break;
24             default:
25                 cout << "Error: " << current_room << endl;
26         }
27     }while(current_room !=3);
28     cout << "You win, Bye!\n";
29 }
30
31 // function definitions (code)
32 int room1(){
33     static int visited = 0; // keeps track of whether this room has been visited
34     already
35     int choice;
36     if(visited == 0){ // if not visited, get a gold star
37         goldstars = goldstars + 1;
38         visited = 1; // change the value so next time it counts as visited
39     }
40     cout << "This is room 1.\n";
41     cout << "1) stay here!\n";
42     cout << "2) go to the subtraction room.\n";
43     cin >> choice;
44     if (choice == 1){
45         return 1; // 'goto' room 1
46     }
47     if (choice == 2){
48         return 2; // goto room 2
49     }
50 }
51 int room2(){
52     int choice;
53     cout << "The subtraction room (2).\n";
54     cout << "1) stay!\n";
55     cout << "2) room 1!\n";
56     cout << "3) exit!\n";
57     cin >> choice;
58     if(choice == 1)
59         return 2; // stay in room 2
60 }
61     if(choice == 2){
62         return 1; // go to room 1
63     }
64     if(choice == 3){
65         return 3; // will exit the game
66 }

```

When we compiled the previous code (prog3.cpp), we got the following error messages:



The screenshot shows the Emacs editor window titled 'emacs@shams'. The file being edited is 'prog3.cpp'. The compilation output shows the following error message:

```

Error message: A statement is written outside function boundaries that start with the word: "if".
Look at line: 61, in the program: "prog3.cpp".
Function room2() ends at line 60 is that what you intend?

```

The error message is displayed in a window titled 'prog3.cpp'. The status bar at the bottom of the window shows 'U***: prog9.txt All L10 (Shell:run Wrap)'.

Figure A- 7 shows page 2 of part 2 of the experiment for group 1 on Qualtrics.

A.4 T-Test for (RQ1: Is there a significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in finding syntax errors?)

$t = \frac{\bar{x}d - \mu}{s/\sqrt{n}} = 4.548825$, where $\bar{x}d = 1.452830189$, $s = 2.325163665$, $df=52$, $t\text{-crit}=1.68$ (one-tail), and $\alpha = 0.05$.

P-value = 1.63E-05.

Table A- 1 shows the results of participants answers for the questions related to RQ1.

Q1) what is the error in the program?										
Q2) in which line is the error?										
Q3) what is the cause of the error.										
Participant #	Group #	Grade for Q1 with GCC/MSVC error messages	Grade for Q2 with GCC/MSVC error messages	Grade for Q3 with GCC/MSVC error messages	X1: Sum of grades of Q1, Q2, Q3 with GCC/MSVC error messages	Grade for Q1 with EduCC error messages	Grade for Q2 with EduCC error messages	Grade for Q3 with EduCC error messages	X2: Sum of grades of Q1, Q2, Q3 with EduCC error messages	Xd=X2 -X1
1	1	2	2	2	6	2	2	2	6	0
2	1	0	0	0	0	2	2	2	6	6
3	1	0	0	0	0	2	2	2	6	6
4	1	0	0	0	0	2	2	2	6	6
5	1	2	2	2	6	2	2	2	6	0
6	3	2	2	2	6	2	2	2	6	0
7	3	2	2	2	6	2	2	2	6	0
8	3	0	2	0	2	2	2	2	6	4
9	3	0	2	2	4	2	2	2	6	2
10	3	2	2	2	6	2	2	2	6	0
11	3	0	2	0	2	2	2	2	6	4
12	5	2	2	2	6	2	2	2	6	0
13	5	0	0	0	0	2	1	2	5	5
14	5	2	2	2	6	2	2	2	6	0
15	5	2	2	2	6	2	2	2	6	0
16	5	2	2	2	6	2	2	2	6	0
17	5	2	2	2	6	2	1	2	5	-1
18	7	0	0	0	0	2	2	1	5	5
19	7	2	2	2	6	2	2	2	6	0
20	7	0	0	0	0	2	2	2	6	6
21	7	1	2	2	5	2	2	2	6	1
22	9	1	2	1	4	1	2	2	5	1
23	9	1	2	2	5	1	2	2	5	0
24	9	1	2	1	4	1	2	2	5	1
25	9	1	2	1	4	0	2	2	4	0
26	11	1	2	2	5	2	2	2	6	1
27	11	0	0	0	0	2	2	2	6	6
28	11	2	2	2	6	2	2	2	6	0
29	11	0	2	0	2	2	2	2	6	4
30	11	2	2	2	6	1	2	2	5	-1
31	11	2	2	2	6	2	2	2	6	0
32	2	0	0	0	0	2	2	2	6	6
33	2	0	0	0	0	2	2	0	4	4
34	2	0	1	0	1	0	2	0	2	1
35	2	2	2	2	6	2	2	2	6	0
36	2	2	2	2	6	2	2	2	6	0
37	4	1	2	1	4	2	2	2	6	2
38	4	0	2	0	2	2	2	2	6	4
39	4	2	2	2	6	2	0	1	3	-3
40	4	2	2	2	6	2	2	2	6	0
41	6	0	0	0	0	2	0	2	4	4
42	6	2	2	2	6	2	2	2	6	0
43	6	2	2	2	6	2	2	2	6	0
44	6	0	2	2	4	2	2	2	6	2
45	6	1	0	2	3	1	0	2	3	0
46	8	2	2	2	6	2	2	2	6	0
47	8	2	2	2	6	2	2	2	6	0
48	8	2	2	2	6	2	2	2	6	0
49	8	2	2	2	6	1	2	2	5	-1
50	12	1	2	2	5	2	2	2	6	1
51	12	2	2	2	6	2	2	2	6	0
52	12	1	1	2	4	1	2	2	5	1
53	12	2	2	2	6	2	2	2	6	0

A.5 T-Test for (RQ2: Is there a significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in fixing syntax errors?)

$t = \frac{\bar{x}_d - \mu}{s/\sqrt{n}} = 3.954749$, where $\bar{x}_d = 0.471698$, $s = 0.868327$, $df=52$, $t\text{-crit}=1.68$ (one-tail), and $\alpha = 0.05$.

P-value = 1.16E-04.

Table A- 2 shows the results of participants answers for the question related to RQ2.

Participant #	Group #	Grade for "How to fix error?" with GCC/MSVC error messages	Grade for "How to fix error?" with EduCC error messages	Xd=X2 -X1
1	1	1	2	1
2	1	0	0	0
3	1	0	2	2
4	1	0	2	2
5	1	2	2	0
6	3	2	2	0
7	3	2	2	0
8	3	2	2	0
9	3	2	2	0
10	3	2	2	0
11	3	0	2	2
12	5	2	2	0
13	5	0	0	0
14	5	2	2	0
15	5	2	2	0
16	5	2	2	0
17	5	2	2	0
18	7	0	2	2
19	7	2	2	0
20	7	0	2	2
21	7	2	2	0
22	9	1	2	1
23	9	2	2	0
24	9	1	2	1
25	9	1	2	1
26	11	2	2	0
27	11	0	2	2
28	11	0	2	2
29	11	0	2	2
30	11	2	2	0
31	11	2	2	0
32	2	0	0	0
33	2	0	1	1
34	2	0	1	1
35	2	2	2	0
36	2	2	2	0
37	4	1	2	1
38	4	0	2	2
39	4	2	2	0
40	4	2	2	0
41	6	0	2	2
42	6	2	2	0
43	6	2	2	0
44	6	2	2	0
45	6	2	2	0
46	8	2	2	0
47	8	2	2	0
48	8	2	0	-2
49	8	2	2	0
50	12	2	2	0
51	12	2	2	0
52	12	2	2	0
53	12	2	2	0

A.6 T-Test (6.3.1 RQ2: Is there a significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in the time-to-find and -fix?)

$t = \frac{\bar{xd} - \mu}{s/\sqrt{n}} = -1.631444786$, where $\bar{xd} = -95.19998113$, $s = 424.8175176$, $df=52$, $t\text{-crit}=-1.68$ (one-tail), and $\alpha = 0.05$. P-value = 5.44E-02.

Table A- 3 shows the results of participants answers for the questions related to RQ3.

Participant #	Group #	X1: Time in seconds for the time participant spent for find and fix questions with GCC/MSVC error messages	X2: Time in seconds for the time participant spent for find and fix questions with EduCC messages	Xd=X2-X1
1	1	166.032	143.357	-22.675
2	1	436.633	735.402	298.769
3	1	213.155	48.405	-164.75
4	1	247.641	151.544	-96.097
5	1	74.988	65.088	-9.9
6	3	37.948	23.222	-14.726
7	3	365.368	124.872	-240.496
8	3	811.175	166.733	-644.442
9	3	898.682	78.823	-819.859
10	3	154.944	143.446	-11.498
11	3	141.992	162.391	20.399
12	5	60.022	96.624	36.602
13	5	291.374	74.524	-216.85
14	5	206.875	173.36	-33.515
15	5	90.339	70.694	-19.645
16	5	142.518	69.833	-72.685
17	5	119.414	87.091	-32.323
18	7	104.079	89.666	-14.413
19	7	211.157	221.815	10.658
20	7	31.241	173.836	142.595
21	7	435.546	943.072	507.526
22	9	232.168	711.978	479.81
23	9	534.229	297	-237.229
25	9	122.681	304.158	181.477
26	9	397.713	93.907	-303.806
27	11	119.222	57.294	-61.928
28	11	526.315	528.826	2.511
29	11	252.921	187.895	-65.026
30	11	115.654	74.915	-40.739
31	11	141.159	107.103	-34.056
32	11	310.498	60.801	-249.697
33	2	572.931	175.203	-397.728
34	2	2187.456	576.406	-1611.05
35	2	2714.305	1246.942	-1467.363
36	2	125.365	259.576	134.211
37	2	797.05	109.868	-687.182
38	4	166.691	462.561	295.87
39	4	754.254	463.727	-290.527
40	4	65.433	53.398	-12.035
41	4	143.86	75.265	-68.595
42	6	864.72	292.405	-572.315
43	6	285.535	83.861	-201.674
44	6	609.157	1532.494	923.337
45	6	496.434	180.595	-315.839
46	6	272.255	77.498	-194.757
47	8	211.125	221.192	10.067
48	8	132.485	810.364	677.879
49	8	114.49	175.117	60.627
50	8	272.622	221.063	-51.559
56	12	170.75	769.494	598.744
57	12	150.156	85.38	-64.776
58	12	110.879	3.267	-107.612
59	12	111.364	134.05	22.686

Appendix B : Source code of the EduCC

B.1 meta.err for parser 1

```

int main()
{}
var ;
:::A statement is written outside function boundaries that start with the word/character:
int main()
{}
cout<<"hi";
:::A print statement is written outside function boundaries that start with the word:
int main()
{}
cin<<"hi";
:::A read statement is written outside function boundaries that start with the word:
int main()
{}
return
::: A statement is written outside function boundaries that start with the word:
void foo()for(int i=0; i<10; i++)
::: Curly bracket "{" is required for the function body, expected "{" before the word:
void foo(){ for(int i=0; i<10; i++)
::: "{" or ";" is expected; if this is a function definition a "{" is required, but if this is a
function declaration a ';' is required before the word/character:
x
int main(){ }
::: this statement is before the boundaries of a function!! token of type VAR
9.1
float main(){ }
::: this statement is before the boundaries of a function!!token of type NUM
for
int main(){ }
:::this statement is before the boundaries of a function!!token of type KEY
int 90 { }
::: this statement neither a function header!! nor a preprocessor directives!! and it is out the
boundaries of a function!!
#include <iostream>
:::is this preprocessor directive, it should be include or define!!
#99 include
::: preprocessor errors, is this preprocessor , the # should be followed by defince or inclue ex.
#include
void foo()cout<<"hello"
::: function header should followed by { and this statment should be part of function boidy inside {}
void foo()"hi"
::: function header should followed by { and this statment/string should be part of function boidy
inside {}
void foo()y=x+9
::: function header should followed by { and this statment should be part of function boidy inside {}
void foo ()
void foo2();
::: function header should followed by ; to be declaration for the function
void foo { }
::: before this token , it should be ; to declare variable or "( )" for function header
int foo(){ x=90+x;
:::the function body need closed curly bracket "}"
int main (){ } }

```

```

::: too many curly bracket
int main() { int x; } y=x+9 }
::: statement (raw text) out (after) the function boundaries
int main() { int x; } y 9+9 }
::: track the state of this case
int foo(){
90
:::statement (number) out (after) the function boundaries
int foo(){}"literal string out the boundaries" ;
::: statement (literal string) out (after) the function boundaries
int foo(){} 9rawtxt
::: statement (raw text-unrecognized)out the function boundaries
"string before the function boundaries"
int main {}
::: part of the function header is missed, ex. int func()
int f()
::: {} body is expected, ex. int func(){...}

```

B.2 meta.err for parser 2

```

if(x>10)||(z<100)
    cout<<"correct answer"<<endl;
:::Is the following operator continuing the condition of a if, for, while, do-while, or switch
statment? I? In that case, you need additional parentheses around the whole condition. The operator is
int x=10, sum=0;
for(x<50)
    sum=sum+x;
:::Semicolons is required in the for header, expected two semicolons inside the for parentheses (i.e
for(int x=1;x<10;x++). You may fix that by adding two ';' in the propoer places: for (statement;
condition ; statment ) before
for(x<50, int i, i++)
    sum=sum+x;
:::Semicolons is required in the for header, expected two semicolons inside the for parentheses(i.e.
for(int x=1;x<10;x++). You may fix that by adding two ';' in the propoer places: for (statement;
condition ; statment ) before
while(i<10) }
cout<<"hi";
:::Do you mean open bracket '{' instead of close bracket '}'? because usually while statement start
with '{' , or any other correct statement before:
if( i<10) }
cout<<"hi";
:::Do you mean open bracket '{' instead of close bracket '}'? because usually if statement start with
'{', or any other correct statement before:
statement start with '{' instead of:
do }
:::Do you mean open bracket '{' instead of close bracket '}'? because usually do-while statement start
with '{', or any other statement(s) before:
do{
sum=sum+x;
cin>>more;
}while(x<10)||(more);
:::Is the following operator continuing the condition of a if, for, while, do-while, or switch
statment? In that case, you need additional parentheses around the whole condition. The operator is
for( i = 0; i < n; i+
    cout << i << ":" <<die << " ";
:::Missing close parenthesis ')' of for header before the word/charater:
do{
sum=sum+x;

```

```

}(x<10);
:::Missing 'while' keyword of the do-while statement before word/character:
for(x==1;;)
  cout<<"hi";
:::Missing the condition part of the for statement, which is required: for( ; condition ; ) before:

```

B.3 yyerror.c for parser 1

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
extern char gfuncName[50];
extern int glastline;
int yyerror_isinitialized, yymaxstate = 39;
struct errtable {
  int i;
  union {
    char *msg;
    struct errtable *p;
  } u;
} errtab[40];
void yyerror_init()
{
  errtab[0].i = 4;
  errtab[0].u.p = (struct errtable *)calloc(1,5 * sizeof(struct errtable));
  errtab[0].u.p[0].u.msg = "this statement is before the boundaries of a function!! token of type
VAR";
  errtab[0].u.p[1].i = 266;
  errtab[0].u.p[1].u.msg = "this statement is before the boundaries of a function!!token of type
KEY";
  errtab[0].u.p[2].i = 265;
  errtab[0].u.p[2].u.msg = "this statement is before the boundaries of a function!!token of type
NUM";
  errtab[0].u.p[3].i = 268;
  errtab[0].u.p[3].u.msg = "part of the function header is missed, ex. int func()";
  errtab[0].u.p[4].i = 262;
  errtab[0].u.p[4].u.msg = "this statement is before the boundaries of a function!! token of type
VAR";
  errtab[4].i = 7;
  errtab[4].u.p = (struct errtable *)calloc(1,8 * sizeof(struct errtable));
  errtab[4].u.p[0].u.msg = "A statement is written outside function boundaries that start with the
word:";
  errtab[4].u.p[1].i = 272;
  errtab[4].u.p[1].u.msg = "A print statement is written outside function boundaries that start with
the word:";
  errtab[4].u.p[2].i = 261;
  errtab[4].u.p[2].u.msg = "too many curly bracket";
  errtab[4].u.p[3].i = 266;
  errtab[4].u.p[3].u.msg = "A statement is written outside function boundaries that start with the
word:";
  errtab[4].u.p[4].i = 271;
  errtab[4].u.p[4].u.msg = "A read statement is written outside function boundaries that start with
the word:";
  errtab[4].u.p[5].i = 265;
  errtab[4].u.p[5].u.msg = "statement (raw text-unrecognized)out the function boundaries";
  errtab[4].u.p[6].i = 268;
  errtab[4].u.p[6].u.msg = "statement (literal string) out (after) the function boundaries";
  errtab[4].u.p[7].i = 262;

```

```

errtab[4].u.p[7].u.msg = "track the state of this case";
errtab[39].i = 1;
errtab[39].u.msg = "the function body need closed curly bracket \"}\"";
errtab[2].i = 2;
errtab[2].u.p = (struct errtable *)calloc(1,3 * sizeof(struct errtable));
errtab[2].u.p[0].u.msg = "is this preprocessor directive, it should be include or define!";
errtab[2].u.p[1].i = 265;
errtab[2].u.p[1].u.msg = "preprocessor errors, is this preprocessor , the # should be followed by
define or include ex. #include";
errtab[2].u.p[2].i = 262;
errtab[2].u.p[2].u.msg = "is this preprocessor directive, it should be include or define!";
errtab[12].i = 1;
errtab[12].u.msg = "before this token , it should be ; to declare variable or \"( )\" for function
header";
errtab[1].i = 1;
errtab[1].u.msg = "this statement neither a function header!! nor a preprocessor directives!! and
it is out the boundaries of a function!!";
errtab[11].i = 7;
errtab[11].u.p = (struct errtable *)calloc(1,8 * sizeof(struct errtable));
errtab[11].u.p[0].u.msg = "Curly bracket \"{\" is required for the function body, expected \"{\"
before the word:";
errtab[11].u.p[1].i = 0;
errtab[11].u.p[1].u.msg = "{} body is expected, ex. int func(){...}";
errtab[11].u.p[2].i = 272;
errtab[11].u.p[2].u.msg = "function header should followed by { and this statment should be part of
function boidy inside {}";
errtab[11].u.p[3].i = 266;
errtab[11].u.p[3].u.msg = "Curly bracket \"{\" is required for the function body, expected \"{\"
before the word:";
errtab[11].u.p[4].i = 258;
errtab[11].u.p[4].u.msg = "\"{\" or \";\" is expected; if this is a function definition a \"{\" is
required, but if this is a function declaration a ';' is required before the word/character:";
errtab[11].u.p[5].i = 263;
errtab[11].u.p[5].u.msg = "function header should followed by ; to be declaration for the
function";
errtab[11].u.p[6].i = 268;
errtab[11].u.p[6].u.msg = "function header should followed by { and this statment/string should be
part of function boidy inside {}";
errtab[11].u.p[7].i = 262;
errtab[11].u.p[7].u.msg = "function header should followed by { and this statment should be part of
function boidy inside {}";
}

int __merr_errors;
extern int yychar;
extern int yylineno;

extern char *yyfilename;

extern char *yytext;

int _yyerror(char *s, int state)
{
    int i;
    char sbuf[128];
    if (!yyerror_isinitialized++) yyerror_init();
    if (strstr(s, "stack")) {fprintf(stderr,"%s", s); return 0;}
}

```

```

if (__merr_errors++ > 10) {
    fprintf(stderr, "too many errors, aborting");
    exit(__merr_errors); }
if (yyfilename) fprintf(stderr, "\nCompilation of (%s):\n", yyfilename);
if ((!strcmp(s, "syntax error") || !strcmp(s, "parse error")) &&
    (state >= 0 && state <= ymaxstate)) {
    if (errtab[state].i == 1)
        s = errtab[state].u.msg;
    else {
        for(i=1; i <= errtab[state].i; i++)
            if(yychar == errtab[state].u.p[i].i) {
                s = errtab[state].u.p[i].u.msg; break;}
        if(i > errtab[state].i && errtab[state].i > 0)
            s = errtab[state].u.p[0].u.msg;
    }
}
if (!strcmp(s, "syntax error") || !strcmp(s, "parse error")){
    sprintf(sbuf, "%s (%d;%d)", s, state, yychar);
    s = sbuf;
}
fprintf(stderr, "\n Error message: %s \"%s\".\n Look at line: %d, in the program: \"%s\".\n
Function %s() ends at line %d is that what you intend?\n\n",
s, yytext, yylineno, yyfilename, gfuncName, glastLine);
return 0;
}

```

B.4 berror.c for parser 2

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int berror_isinitialized, bmaxstate = 88;
struct berntable {
    int i;
    union {
        char *msg;
        struct berntable *p;
    } u;
} berntab[89];
void berror_init()
{
    berntab[15].i = 2;
    berntab[15].u.p = (struct berntable *)calloc(1, 3 * sizeof(struct berntable));
    berntab[15].u.p[0].u.msg = "Is the following operator continuing the condition of an if, for,
while, do-while, or switch statement? In that case, you need additional parentheses around the whole
condition. The operator is";
    berntab[15].u.p[1].i = 267;
    berntab[15].u.p[1].u.msg = "Do you mean open bracket \xe2\x80\x98{\xe2\x80\x98 instead of close
bracket '}'? because usually if statement start with \xe2\x80\x98{\xe2\x80\x98 , or any other correct
statement before:";
    berntab[15].u.p[2].i = 273;
    berntab[15].u.p[2].u.msg = "Is the following operator continuing the condition of an if, for,
while, do-while, or switch statment? In that case, you need additional parentheses around the whole
condition. The operator is";
    berntab[62].i = 1;
}

```

```

    berrtab[62].u.msg = "Do you mean open bracket \xe2\x80\x98{\xe2\x80\x98 instead of close bracket
'}'? because usually while statement start with \xe2\x80\x98{\xe2\x80\x98 , or any other correct
statement before:";
    berrtab[88].i = 1;
    berrtab[88].u.msg = "Is the following operator continuing the condition of an if, for, while, do-
while, or switch statement? In that case, you need additional parentheses around the whole condition.
The operator is";
    berrtab[50].i = 2;
    berrtab[50].u.p = (struct berrtable *)calloc(1,3 * sizeof(struct berrtable));
    berrtab[50].u.p[0].u.msg = "Semicolons is required in the for header, expected two semicolons
inside the for's parentheses (i.e for(int x=1;x<10;x++). You may fix that by adding two ';' in the
propoer places: for (statement; condition ; statement ) before ";
    berrtab[50].u.p[2].i = 265;
    berrtab[50].u.p[2].u.msg = "Semicolons is required in the for header, expected two semicolons
inside the for's parentheses (i.e for(int x=1;x<10;x++). You may fix that by adding two ';' in the
propoer places: for (statement; condition ; statement ) before ";
    berrtab[50].u.p[3].i = 275;
    berrtab[50].u.p[3].u.msg = "Semicolons is required in the for header, expected two semicolons
inside the for's parentheses(i.e. for(int x=1;x<10;x++). You may fix that by adding two ';' in the
propoer places: for (statement; condition ; statement ) before ";
    berrtab[33].i = 1;
    berrtab[33].u.msg = "Do you mean open bracket \xe2\x80\x98{\xe2\x80\x98 instead of close bracket
'}'? because usually do-while statement start with \xe2\x80\x98{\xe2\x80\x98 , or any other
statement(s) before:";
    berrtab[80].i = 1;
    berrtab[80].u.msg = "Missing close parenthesis ')' of for header before the word/charater:";
    berrtab[64].i = 1;
    berrtab[64].u.msg = "Missing 'while' keyword of the do-while statement before word/character:";
    berrtab[63].i = 1;
    berrtab[63].u.msg = "Missing the condition part of the for statement, which is required: for( ;
condition ; ) before:";
}

int __mberr_errors;
extern int bchar;
extern int blineno;

extern char *bfilename;

extern char *btext;

int _berror(char *s, int state)
{
    int i;
    char sbuf[128];
    if (! berror_isinitalialized++) berror_init();
    if (strstr(s, "stack")) {fprintf(stderr,"%s", s); return 0;}
    if (__mberr_errors++ > 10) {
        fprintf(stderr, "too many errors, aborting");
        exit(__mberr_errors); }
    if (bfilename) fprintf(stderr, "\nCompilation of: (%s)\n", bfilename);
    if ((!strcmp(s, "syntax error") || !strcmp(s,"parse error"))&&
        (state>=0 && state<=bmaxstate)) {
        if (berrtab[state].i==1)
            s = berrtab[state].u.msg;
        else {
            for(i=1;i<=berrtab[state].i;i++)

```



```

        if(bchar == berrtab[state].u.p[i].i) {
            s=berrtab[state].u.p[i].u.msg;break;}
        if(i>berrtab[state].i && berrtab[state].i > 0)
            s=berrtab[state].u.p[0].u.msg;
    }
}
if (!strcmp(s, "syntax error") || !strcmp(s, "parse error")){
    sprintf(sbuf,"%s (%d;%d)", s, state, bchar);
    s=sbuf;
}
fprintf(stderr, "\n Error message: look at line %d, in the program \"%s\".\n %s \"%s\".\n
\n",blineno,bfilename, s,btext);

return 0;
}

```

B.5 main.c

```

#include <stdio.h>
#include <stdlib.h>
#include<string.h>
#include<unistd.h>
#include "p.h"
int yyparse();
char *yyfilename;
char *body="";
char *filename;
char *bfilename;
struct tree *root;
struct tree *branch;
struct tree *b_root;
struct branch_list *head_branch_list;
struct tree *b;
extern FILE *yyin;
extern int yylineno;
extern int yydebug;
extern int bdebug;
int bwrap()
{
    return (1);
}
char *mybuff;
int main(int argc, char *argv[])
{
    mybuff=malloc(5000);
    int i;
    if (argc<2) {
        fprintf(stderr, "usage: sample file\n");
    }
    if ((yyin = fopen(argv[1],"r")) == NULL) {
        fprintf(stderr, "no %s\n", argv[1]); exit(1);
    }
    yyfilename=argv[1];
    bfilename=argv[1];
    // printf("\nfunctions analysis by parser one\n");
    i = yyparse();

    // treeprint(root,1);

```

```
// printf("yyaparse returned %d\n", i);  
if(i==0)  
    parser2(root);  
  
return 0;  
}
```