Techniques for Enhancing Compiler Error Messages

A Proposal
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
Sana’a 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.

December 2021
Abstract

Teaching and learning introductory programming in the first year of colleges is one of the important challenges that face computer science practitioners. Knowing when and how to give feedback to learners in their code is a vital question in this research area and this implies we must overcome the difficulties of compiler error messages. This study proposes a model for integrated learning development environment and presents an analysis of the syntax error messages from the learner’s perspective. It also proposes a technique to parse the source code in three phases that may help in generating better syntax error messages. Finally, it proposes an experimental design to evaluate the proposed parsing technique whether it enhances the quality of syntax error messages for the learners or not.
# Table of Contents

Abstract ........................................................................................................................................... ii

Chapter 1: Introduction .................................................................................................................. 2

Chapter 2: Background .................................................................................................................. 4

2.1 Compiler error messages are considered unhelpful ................................................................. 4

2.2 How should the compiler reports the syntax errors ............................................................... 4

2.3 Can we overcome the limitation of the parser in generating better error messages? .......... 6

Chapter 3: Integrated Learning Development Environment for Learning and Teaching C/C++
Language to Novice Programmers ................................................................................................. 8

2.1 Introduction .............................................................................................................................. 8

2.2 Challenges Facing Novice Programmers .............................................................................. 9

2.2.1 Misconception .................................................................................................................... 9

2.2.2 Error handling and debugging .......................................................................................... 10

2.3 ILDE ......................................................................................................................................... 11

2.4 Evaluation .............................................................................................................................. 16

2.5 Summary .................................................................................................................................. 16

Chapter 4: Analysis of Syntax Error Messages from Learner Perpective .................................. 18

Chapter 5: Reengineering the Compiler ..................................................................................... 30

Chapter 6: Experiment in the classroom ...................................................................................... 32
Chapter 1: Introduction

Programming is a required competency in the journey of computer science and students in related fields. Some students start learning programming in high school or earlier. They need to have skills of converting real life problems into computer solutions. They need to understand the notional machine, the syntax and semantics of the programming languages, to develop problem-solving skills, to design solutions, to write code, and to debug their code. Many college CS curricula persist in beginning with C++ or Java in their first courses for majors, and with good reason. But their relative difficulty makes the need for an educational customized compiler greater, rather than lesser than is needed 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.

Educational development environments have been developed to help in teaching programming to beginners of different ages, some of them for kids such as GreenFoot, some for high school students such as TigerJython, and some for college level students such as BlueJ. Most of these educational environments are for Java and Python. We propose an educational development environment for C/C++ and we present it in chapter two. In chapter three we analyze the quality of syntax error messages. In chapter four we propose a parsing technique that may help in generating better error messages. In chapter five we propose an experimental design that evaluates the quality of syntax error messages that are generated by a compiler implementing these proposed techniques.

Dissertation Objectives. The objective of this research is to enhance compiler error messages, so they are more helpful to learners. In order to do this, we propose to design a compiler that overcomes the limitation of how parsers report errors. In addition, the compiler will supply different error messages based on the learners’ level and that follows the guidance from HCI and CS education literature.

To achieve this objective, the following will be done:

- Investigate the challenges that face novice programmers
- Find a solution to integrate learning and feedback into the development environment.
- Identify the common errors of novice programmers from the literature.
- Investigate the quality of syntax error messages of mainstream compilers.
- Find 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.
Research questions: 1) Can modified parsing techniques help in generating better syntax error messages? 2) Can the compiler writers help in writing learning feedback that is customized to different students’ levels?

Methodology. We searched for common errors of novice programmers in the literature, for C++ language and languages from the same family such as Java. We have analyzed how popular compilers are reporting these common errors. Popular compilers that are used by students in the introductory programming courses. We also planned an experimental design to test a proposed parsing technique that we think may enhance the quality syntax error messages.

Terminology. Syntax error: the term syntax error in this proposal refers to a parsing failure during the validation 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 is a message that is generated by a compiler to the user reporting a syntax error.

Quality of syntax error message: this definition is inspired by software engineering definition of quality [1].

Bad quality: the syntax error message does not deliver what the average programmer expects the message to have. “For example, customers expect a car to have four tires, a windshield, windshield wipers, and a steering wheel”. So, if the message is incorrect or not understandable it is a bad message. This level of quality confuses and misleads the programmer, 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.
Chapter 2: Background and Related Works

A compiler is a communication tool; one of its primary roles is to detect errors and report detected 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” [1]. For programmers, especially novices, it is important that the compiler provides helpful error messages 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 considered 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 of builds over months 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 [2].

In 2019, a survey conducted by Becker et al showed that both current and old research describe 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 [3].

2.2 How should the compiler reports the syntax errors

The research continues to study and discuss the quality of compiler error messages. In 1982, Shneiderman [1] studied the impact of error messages on the users. 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) avoiding the
negative terms such as “illegal”, “invalid”, “error”, or “incorrect”. 5) avoid obscure terms such as “syntax error”. 5) be comprehensible.

In 1983, Brown [2] conducted an experiment to test the quality of error messages. He analyzed the error messages of fifteen Pascal compilers for a group of simple and common erroneous programs. He found that the majority of compilers in the experiment were ranging from “barely accepted” to “laughable”. Brown recommended that error messages should be friendly, give an informative message, and provide help in correcting the error. He also recommended 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 [3] analyzed the usefulness of error and warning messages for different user levels and received the feedback from users for years and improved the compiler error messages accordingly. The compiler was locally built for Modula-2 and used by students over a number of years in their department. Although they didn’t measure the effectiveness of improved compiler warning and error messages, they developed a group of merits for the useful compiler warning and error messages: 1) the compiler error message needed to be helpful 2) the compiler hints at how to fix the problem 3) the compiler should take into consideration different user levels, for example user can use -students flag for more detailed error checks. The compiler also provides extra warnings appropriate for first year students. These warnings include checking for identifiers which look like keywords, same name variables in different visible scopes, variables and parameters declared but never used and variables used before being initialized. 4) compiler in some cases provide a description of what the compiler believes it has seen.

In 2010, Traver [4] proposed a set of principles that should guide compiler error messages design. These principles are: 1) clarity and brevity 2) specificity 3) context insensitivity 4) locality 5) proper phrasing 6) consistency 7) suitable visual design 8) extensible help. Traver discussed the problem of the context-sensitivity of errors; that the compiler gives different error messages for the same error.

In 2011, Marceau et al. [5] investigated the effectiveness of error messages for the DrScheme environment. DrScheme is designed for students to teach and learn 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 edit different kinds of errors, interviewed and quizzed the students. They found that the metrics of a rubric that measure the effectiveness of good error messages are: 1) students can read. 2) students can understand. 3) students can formulate a response for the error. 4)
students can fix the error. Also, they found that using technical vocabularies at the error message make it unclear instead of helpful. Even the programming terminologies that students hadn’t studied before confused them, such as “function body” for beginner users. Marceau et al. found that students prefer the hint on how to fix the error more than the highlight of the error area in the source code.

In 2018, Barik et al. [6] postulated that the quality of compiler error messages improved if they contain explanations; the compiler error message should apply explanation theories, such as Toulmin’s model of argument. 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 they asked professional developers from software companies to make preference for selected error messages from OpenJDK, and Jikes. Where some messages are following the reasoning model, and some are deficient. 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 the explanation model over the deficient one, 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 tool, such as Error Prone implements 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. A study by Barik et al. [7] found that programmers spent up to 25% of their task time on reading error messages. The study used eye-tracking research with programmers. Another study by Becker et al. [8] 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.

2.3 Can we overcome the limitation of the parser in generating better error messages?
Hirsotova et al. [8] developed a preprocessor tool called Expresso to detect common Java errors in the novice Java codes and report friendly and better error messages with hints on how to fix them. 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 [9] developed a parser for Python programs capable of recognizing a group of error patterns. 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 misconceptions problem and he found part of the problem is that students project the mathematical mental model when writing the code. So, they write incorrect expressions in Python programs. He implemented this parser as part of the successful educational environment (TigerJython) that delivers error messages in German language. The limitation of their parser is that not all novice errors derive from the mathematical mental model.

Jeffery’s approach [10] which is applicable for different programming languages, is an open source code tool called Merr. Merr takes additional information from the parsers generated by yacc and bison family and passes them to the compiler error message system. More accurately, Merr generates automatically the code of the error reporting function in the compiler (e.g. yyerror()). It provides the state that the error occurs on, and the erroneous token. The downside of Merr approach is that the compiler writer must come up with erroneous fragments inorder to associate messages with the states of the parser where errors can occur.

Pottier [11] argued that an LR parser can generate good diagnostic messages using Jeffery’s approach. He elaborates on it by designing an algorithm that can generate the erroneous fragments automatically. Also, he proposed three features of a good diagnostic message that is mapped from the erroneous fragments by the Merr tool: correctness “This collection of erroneous input sentences should ideally be correct (i.e., every sentence is erroneous), irredundant (i.e., no two sentences lead to the same state), and complete (i.e., every state where an error can occur is reached by some sentence)” [11].
Chapter 3: Integrated Learning Development Environment for Learning and Teaching C/C++ Language to Novice Programmers

This chapter is adopted from:


This chapter presents an Integrated Learning Development Environment (ILDE) that integrates technologies with pedagogies for first-year students learning to program. 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.

2.1 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 [1]. Much like Wonderland, learning how to program feels like a multidimensional journey filled with different perspectives and tasks.

This paper presents an Integrated Learning Development Environment (ILDE) to enhance the learning and teaching of programming for first- and second-year computer science students. ILDE integrates technologies with pedagogies. The design of the ILDE model is built upon Cognitive Load Theory [2],[3], Kolb's Experiential Learning Theory [4], Constructivism principles [5], Cognitive Theory of Multimedia Learning [6], 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 post-secondary 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 hypotheses: 1) ILDE's Multimedia Learning Content deploys visualization of computation to enhance novices' performance and 2) ILDE's Informative Feedback that utilizes a customized compiler will enhance 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.

2.2 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 [7] 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 program code to solve a problem similar to the problems already learned; create program code to solve new problems. The learner must also understand the machine operations that a programming language expresses [8]. 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 [8][9]-[10].

2.2.1 Misconception
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 [8], [10], [11], [12], [13], [14]. 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. Fig. 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 builds 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 utilizes 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 plays a video that explains the memory allocation process and how uninitialized memory affects program
logic. For learners at the intermediate level, the feedback subsystem invokes 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[15]. On the other hand, it is very important that the learner understands the ‘inner world’ of the programming language being used [8],[15]. 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; the relation between task structure and programming language semantics [16]. 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.

2.2.2 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 [11], [17]. 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 [18][19]-[20].

A logic error is when the produced program does not perform the intended functionality, for example due to improper converting between data types, or incorrect division of a float number by an integer [11], [17]. Ettles, Luxton-Reilly, and Denny report that misconceptions are the source of logic errors [12]. Finding and fixing logical errors is more difficult than fixing syntax errors. Debugging skills are difficult and even good programmers often lack these skills [12].

Many tools have been integrated into IDEs to enhance messages for both syntax and logic errors. Specialized IDEs have been created for education purposes. These efforts include PROUST[22], Merr[23], Expresso[11], BlueJ [24], TigerJython [13], Alice [25], Scratch [25], Greenfoot [25], WebTigerJython [26]. These projects are focused efforts targeting specialized aspects of programming instruction. Thus, novices still need assistance mastering basic concepts.
ILDE is designed to be used in the introductory programming courses where it focuses on the basic programming concepts and offers informative feedback. 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[15], [16], [27].

2.3 ILDE
ILDE draws upon Cognitive Load Theory, Kolb’s Experiential Learning Theory, Constructivism principles, and Cognitive Theory of Multimedia Learning (CTML) [2][3][4][5]-[6]. 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 CTML, 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 [28].

**Multimedia Learning Content with Visualization Tool.**

The 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 similar to 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 where to write, what button to press to compile, and so on. The task text will be available as an image dockable 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 style 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 a 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, a main concept of programming. The lesson starts with a video explaining the iteration structure in the C language. It shows the anatomy of the for control structure. For example, the anatomy of the statement

\[\text{for}\(i=1; i <= 7; i++\) \text{ weekly_product += daily_product;}\]

\[\text{can be illustrated by:}\]

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 a 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 a 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. Fig. 2 and Fig. 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.

Figure 2.2 is an 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 2.3 is an 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 in a textbox, and the design of the problem in a textbox, 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 enable 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.

2.4 Evaluation
We will evaluate the ILDE model by experiments during learning to assess the effects of the ILDE's feedback and learning subsystems on students' performance. We will design and implement two different experiments.

The first experiment will test the first hypothesis: "ILDE's Multimedia Learning Content can deploy visualization of computation to enhance novices' performance". The setting of the experiment will involve enabling ILDE's MLC features and disabling the ILDE's feedback subsystem features and compare the effects of this situation with regular IDE, on novice performances' acquisition of skills.

The second experiment will test the second hypothesis: "ILDE's Informative Feedback that utilizes a customized compiler will enhance novices' performance". The setting of the experiment will involve enabling ILDE's feedback subsystem features and disabling the ILDE's MLC feature and compare the effects of this situation with regular IDE on novice performances' acquisition of skills.

Both experiments will follow a quasi-experimental, pre-test/post-test design. Participants of the experiments will be undergraduate students of computer programming classes at the computer science department in a public research university.

2.5 Summary
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. Many college CS curricula persist in beginning with these languages in their first courses for majors, and with good
reason. But their relative difficulty makes the need for an educational learning development environment greater, rather than lesser than is needed in "easier" first languages.

The next steps involve continuing to work on the proposed technical solutions of the customized compiler and MLC. Then, we will finish the implementation with plans to test ILDE next academic year, 2020-2021.

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 Learner Perpective

To explore the problem of unhelpful syntax error messages, let us examine the behaviour of two popular compilers: GNU GCC version 10.3.1 (GCC) and Microsoft Visual C++ version 2019 (VC++), on a set of erroneous programs that we present in Figure 4.1, Figure 4.7, and Figure 4.12. Consider prog2.cpp from Figure 4.1. 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 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.

GCC and VC++ report the error of prog1.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 says: “expected a statement “ in line 7, “this declaration has no storage class or type specifier” in line 8, “ expecte a ‘;’ in line 8. The content of the messages are incorrect and aren't understandable. They use compiler writers’ jargon such as “primary-expression”, and “storage class or type specifier”.

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.
Figure 4.1 shows C++ programs with common syntax errors of novice programmers, unbalanced curly brackets.
GNU GCC report:

```
[sanaadmin@shams ErrExamples3]$ g++ prog2.cpp
prog2.cpp: In function 'int recursive(int)'
  prog2.cpp:10:1: error: expected ')' at end of input
  10 |   }
     | ^
  prog2.cpp:5:22: note: to match this '{'
     5 |   int recursive( int x)^
```

Microsoft Visual C++ report:

![Error List](image)

Figure 4.2 shows good 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++ prog1.cpp
prog1.cpp: In function 'int main()'
  prog1.cpp:7:3: error: expected primary-expression before '}' token
     7 |   }
```

Microsoft Visual C++ report:

![Error List](image)

Figure 4.3 shows bad 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++ prog3.cpp
 prog3.cpp:10:15: error: expected unqualified-id before ‘)’ token
   res=f(x)+f(y);
   ^
 prog3.cpp:11:13: error: unqualified-id before ‘++’ token
   ++res;
   ^
 prog3.cpp:12:3: error: ‘cout’ does not name a type
   cout<"result="<res;
   ^~~~
 prog3.cpp:13:1: error: unqualified-id before ‘)’ token
   )
```

Microsoft Visual C++ report:

![Error List](image)

Figure 4.4 shows 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:12:13: error: expected unqualified-id before ‘+’ token
     res=f(x)+f(y);
     ^
```

```
[1416093323] Error 1
```

```
[sanaadmin@shams ErrExamples3] g++ prog4.cpp
```

```
[1416093323] Error 1
```
Analyzing the previous examples from a parsing techniques perspective, we will find that the quality of error messages are 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 prog1.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 ‘;’”. But line 8 itself has no syntax error if it is placed inside the boundaries of a function body. While for prog2.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. We considered that GCC’s message is a good message for prog2.cpp, although it contains a compiler writer jargon “primary-expression”, and this sometimes could be a distraction even if the message is correct. But at this point we will focus on how a parser recognizes the source code and how they report the errors accordingly. We can conclude at this point that 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 errors.
Figure 4.6 shows again the C++ programs in Figure 4.1, with drawings that clarify how a parser recognizes the function body boundaries.

Studying GCC and VC++ reports for syntax errors in the control structures, we find 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 at how confusing and misleading these messages are, despite most of them only having this single error. It is worse when there is an error or more in the expression inside the header parentheses besides missing one or both its parentheses.
Figure 4.7 shows 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]$ g++ prog5.cpp
```

Microsoft Visual C++ report:

![Error List](image)

Figure 4.8 shows 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: 1 value required as unary '& operator
    7 | if(a>b<=200)&&(b<200)
    | ^

[sanaadmin@shams ErrExTest]$ g++ prog6.cpp
```

Microsoft Visual C++ report:

![Error List](image)

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:

```
$ gcc prog7.cpp
prog7.cpp: In function ‘int main()’:  
prog7.cpp:7:15: error: expected primary-expression before ‘||’ token
    if(a*b>=200) || (b<200)
    ^
[...]
```

Microsoft Visual C++ report:

![Error List Table]

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:

```
$ gcc prog8.cpp
prog8.cpp:10:6: warning: init-statement in selection statements only
7) or -std=gnu++1z
10) if (var3 < (var2 & var3) ^
    cout << "The smallest number is " << var1;

prog8.cpp:12:11: error: expected primary-expression before ‘}’ token
12 })
    cout << "The smallest number is " << var1;

prog8.cpp:12:11: error: expected primary-expression before ‘}’ token
12 }
```

Microsoft Visual C++ report:

![Error List Table]

Figure 4.11: Bad quality syntax error messages reported by GCC and VC++ on compilation of prog8.cpp in Figure 4.7
Loop control structures, especially for statement, are problematic to learners. Many misconceptions 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 the compilers’ behaviours 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 confusing ways to the learners. Notice how both generate lists of incorrect error messages in Figure 4.16 for the prog12.cpp.

Figure 4. 12: Shows 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  
          for(x+=0 ; y<10)  
               ^
 prog9.cpp:10:1: error: expected primary-expression before ‘}’ token  
            } 
            ^
 prog9.cpp:9:3: error: expected ‘;’ before ‘}’ token  
         }  
         ^
 prog9.cpp:10:1: error: expected primary-expression before ‘}’ token  
            } 
            ^
 prog9.cpp:9:3: error: expected ‘)’ before ‘}’ token  
         )  
         ^
 prog9.cpp:6:5: note: to match this ‘(’  
          for(x+=0 ; y<10)  
               ^
 prog9.cpp:10:1: error: expected primary-expression before ‘}’ token  
            } 
            ^
 [sanaadmin@shams ErrExTest]$
```

Microsoft Visual C++ report:

![Error List Table]

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
  for(x=0, x<9,x++)
   ^

 prog10.cpp:9:20: error: expected ‘)’ before ‘;’ token
  cout<<"sum="<<sum;
  ^

 prog10.cpp:6:6: note: to match this ‘(’
  for(x=0, x<9,x++)
  ^

[sanaadmin@shams ErrExTest]$

Microsoft Visual C++ report:

![Error List]

Figure 4.14: Bad quality syntax error messages reported by GCC and VC++ on compilation of prog10.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
  for(x=0, x<9,x+++)
   ^

 prog11.cpp:9:20: error: expected ‘)’ before ‘;’ token
  cout<<"sum="<<sum;
  ^

 prog11.cpp:6:6: note: to match this ‘(’
  for(x=0, x<9,x+++)
  ^

[sanaadmin@shams ErrExTest]$

Microsoft Visual C++ report:

![Error List]

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@shans 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
   |    cin>>y;
   |    //
   |    std::istream {aka std::basic_istream<char>}
 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_iterator.h:67,
  from /usr/include/c++/10/bits/char_traits.h:39,
  from /usr/include/c++/10/ios:40,
  from /usr/include/c++/10/iostream:38,
  from /usr/include/c++/10/ios:39,
  from prog12.cpp:1:
/usr/include/c++/10/bits/stl_iterator.h:533:5: note: candidate: ‘template<typename _Iterator> std::reverse_iterator<_Iterator> std::operator+(typename std::reverse_iterator<_Iterator>::difference_type, const std::reverse_iterator<_Iterator>&)’
     533 | std::operator+(typename reverse_iterator<_Iterator>::difference_type, const
     | operator+(typename reverse_iterator<_Iterator>::difference_type, const
     | n, 1
     | ^

/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:

![Error List](image)

Figure 4.16: Bad quality syntax error messages reported by GCC and VC++ on compilation of prog12.cpp in Figure 4.12
Chapter 5: Reengineering the Compiler

For this dissertation we propose a technique that parse the source code into three phases. Phase one analyzes the source code into its major constituent parts, functions. Phase two analyzes control structures that have been found inside the functions’ bodies in phase one. Phase three analyzes the fine-grained statements and expressions of the source code. Each phase does lexical analysis, syntax analysis, and generates abstract syntax trees. Also, it uses Merr [ref] with each phase parser, to automatically produce a mapping of parse states to diagnostic messages, so each parser has its own error reporting function.

Phase one analyzes the outer skeletal structure of the functions in the source code. Phase one decides whether the source code conforms to the rules for function declaration and function definition. The lexical analysis of this phase recognizes only the tokens that are related to the function declaration and definition. The syntax analysis of this phase validates the rules that are 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, location, and pointers to children. In Parser one, the first child of a function is the header part, and the second child is the body part. These children are of type string, and they are not validated at this phase for any rules. The next phase analyzes the second child, body, and decides if it follows the rules for the control structures.

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 that are related to the iterations, if-statement, switch structures. Also, the syntax analysis of this phase validates the rules that are related to the iterations, if-statement, switch structures. It reports errors using a 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 parts and body parts that are processed in phase one and phase two.
This phase analyzes the statements that are found in the functions’ headers and bodies, and control structures' headers and bodies, and the statements found outside the function boundaries. These statements are aggregated into one string buffer and are notated with their parents’ information. It reports errors using the Merr tool and has its yyerror_d() and generates an abstract syntax tree (AST-d).
Chapter 6: Experiment in the classroom

We will evaluate the quality of syntax error messages of EduCC with an experiment. The purpose of this experiment will be to compare the quality of syntax error messages of EduCC with mainstream compilers: GNU GCC and Microsoft Visual C++ (VC++) for learners. Participants will debug a set of erroneous C++ programs. The independent variable will be the type of compiler (EduCC, G++, VC++), the dependent variable is the quality of syntax error messages, and it will be measured by the occurrence of found-error, occurrence of fixed-error, mean-time-to-find, and mean-time-to-fix.

The participants will be the undergraduate students of engineering college at the University of Idaho who will be studying computer science I (CS120). The expected number of participants is between 80 and 100. Independent variable will be type of compiler (EduCC, G++, VC++). Dependent variables will be the surrogate of the quality of syntax error messages of each mentioned compiler: Occurrence of found-error, Occurrence of fixed-error, Mean-time-to-find, Mean-time-to-fix. The Instruments will be set of erroneous programs. We will prepare a set of C++ programs that will be seeded with the common errors. We will choose a set of programs that are part of the programming exercises (assignments and labs) that students are going to study next semester, to exclude the understanding of the exercises.

The Null hypotheses for the experiment will be:

Hypothesis 1: There is no significant difference between the quality of syntax error messages that generated by EduCC, GNU G++, and Microsoft Visual C++ in finding syntax errors.

Hypothesis 2: There is no significant difference between the quality of syntax error messages that generated by EduCC, GNU G++, and Microsoft Visual C++ in fixing syntax errors.

Hypothesis 3: There is no significant difference between the quality of syntax error messages that generated by EduCC, GNU G++, and Microsoft Visual C++ in the mean-time-to-find.

Hypothesis 4: There is no significant difference between the quality of syntax error messages that generated by EduCC, GNU G++, and Microsoft Visual C++ in the mean-time-to-fix.

The alternative hypotheses for the experiment will be:

Hypothesis 1: There is significant difference between the quality of syntax error messages that generated by EduCC, GNU G++, and Microsoft Visual C++ in finding syntax errors.
Hypothesis 2: There is significant difference between the quality of syntax error messages that generated by EduCC, GNU G++, and Microsoft Visual C++ in fixing syntax errors.

Hypothesis 3: There is significant difference between the quality of syntax error messages that generated by EduCC, GNU G++, and Microsoft Visual C++ in the mean-time-to-find.

Hypothesis 4: There is significant difference between the quality of syntax error messages that generated by EduCC, GNU G++, and Microsoft Visual C++ in the mean-time-to-fix.

Chapter 7 (1 page, 2 page) Conclusion

Summary, restate what you hope to learn, find answer to research question, ... in this dissertation and short description what schedule,
References


