A Parsing Technique for Enhancing Compiler Syntax Error Messages for Student Programmers

Sana Algaraibeh
Computer Science and Engineering
New Mexico Institute of Mining and Technology
Socorro, NM, USA
sanaa.algaraibeh@nmt.edu

Clinton Jeffery
Computer Science and Engineering
New Mexico Institute of Mining and Technology
Socorro, NM, USA
clinton.jeffery@nmt.edu

Terence Soule
Computer Science
University of Idaho
Moscow, ID, USA
tsoule@uidaho.edu

Tonia Dousay
School of Education
University of Alaska Anchorage
Anchorage, AK, USA
tadousay@alaska.edu

Abstract*—Contribution: This full research paper presents an innovative parsing technique that aims to improve syntax error messages for undergraduate students. The quality of syntax error messages generated by the new parsing technique was evaluated and compared with the messages produced by mainstream compilers. Background: Unfortunately, compiler error messages are often unhelpful. The study explains some intrinsic challenges faced in generating good syntax error messages and presents a global, local, and expression-level (GLE) parsing technique to overcome some of these challenges. GLE is a 3-phase parsing that 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. Research Question: Can GLE parsing techniques help generate better syntax error messages? Methodology: The study evaluated the quality of syntax error messages generated by the proposed GLE parsing technique. The evaluation was done in a controlled experiment and within-group design where participants found and fixed errors in erroneous programs using accompanying error messages from different compilers. The independent variable is the compiler type. 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. Three questions were used to evaluate the "helping in finding errors" quality of the error message: 1) what is the error in the program? 2) in which line is the error? 3) what is the cause of the error? One question was used to evaluate the quality of "helping in fixing errors": "how to fix the error?" The time that participants used to find and fix a program was calculated. The participants were 51 undergraduate students in the Computer Science and Engineering department at the New Mexico Institute of Mining and Technology. Findings: The results show there is a significant statistical difference in finding errors and fixing erroneous programs using messages generated by the proposed GLE parsing technique and two mainstream compilers: GNU GCC and Microsoft Visual C++. No significant difference exists in the time-to-find and -fix. The result indicates that the proposed GLE parsing technique can help generate better error messages for undergraduate students.

***Index Terms*—Novice programmers, syntax error messages, compiler error messages, parsing technique, and measuring the quality of syntax error messages.**

I. Introduction

The engineering education community is committed to improving interactive learning environments, promoting independent learning and practice, and helping students master essential field-specific skills. Feedback plays a crucial role in students' success [1]. Programming courses are vital in engineering education, especially for computer science students. Various integrated development environments (IDEs) such as BlueJ [2] and TigerJython [3] have been created to support learners. An IDE is a computer environment that allows programmers to write, compile, debug, and run source code. In a previous study, we introduced an integrated learning development environment (ILDE) that leverages compiler error messages to provide constructive feedback to learners [4]. However, unhelpful compiler error messages remain a significant obstacle in developing IDEs.

What do we mean by unhelpful error messages? When students write source code in high-level languages such as C/C++, Java, or Python, they run a compiler to convert the code into machine language. Normally, the source code will contain errors, and when this happens, the compiler will not convert the source code into machine language but will report errors in the source code. These error reports, also known as compiler error messages, are meant to help programmers find and fix errors. However, the error messages generated by mainstream compilers can be incorrect, misleading, and confusing, particularly for novice programmers. This can hinder the learning process and prevent effective feedback.

The parser, a compiler component, is responsible for generating syntax error messages. Syntax errors occur due to misspelled, missing, or incorrectly ordered words in the source code. Despite receiving significant attention from researchers and practitioners over the years, writing good compiler error messages is still difficult. Some researchers are looking to compiler writers for an explanation. Should compiler writers be blamed for the poor error messages? Compiler development is an active field with ongoing advancements in technology. Compiler writers primarily focus on the efficiency of software in terms of time, memory, and production. For example, while the LR parser is renowned for its efficiency, it also

has limitations in providing useful error messages. This study delves into the limitations of the LR parser and proposes an improvement.

The article is organized as follows: Section II presents related literature. Section III discusses why LR parsers sometimes produce poor error messages. Section IV proposes a solution to address specific limitations of LR parsers. Section V evaluates the proposed solution. Section VI presents results, while Section VII outlines limitations. Lastly, Sections VIII and IX provide a discussion and conclusion.

II. Related Literature

Novice Programmers often struggle to understand and utilize compiler error messages. In an eye-tracking study, Barik et al. found that reading error messages is a cognitively demanding task. They compared the difficulty and time college students spent reading source code to reading compiler error messages for incorrect Java programs. They found that students spent substantial time reading error messages, 13%-25% of total task time. Also, they found that as revisits to error messages increase, the probability of successfully resolving a compiler error decreases [5].

Denny and his colleagues, in their research, found that students spent most of the time fixing common syntax errors such as "missing ;", "missing {", "missing }", or "missing)". Denny et al.'s findings align with the results of Jadud, Jackson et al., and Hristova et al. [6]–[9].

Poor error messages not only hinder learners but also negatively impact the productivity of expert programmers at software companies. Seo and colleagues conducted a comprehensive study at Google to examine how programmers in the industry interact with their compilers and build tools. The study involved analyzing 26.6 million builds of software written in C++ and Java languages and included 18,000 developers over a period of nine months. The results revealed that 37.4% of C++ builds and 29.7% of Java builds fail. Also, they found that the errors with the highest frequencies are common and simple, such as expected right parenthesis and the use of undeclared variables. [10].

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

Hristova et al. developed a preprocessor tool called Expresso to detect common Java errors in novice Java codes and report friendly and better error messages with hints on how to fix them [9]. 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 [3]. 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 found that part of the problem is that students project a mathematical mental model when writing code. So, they write incorrect expressions in Python programs. He implemented a 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 [11]. Merr takes additional information from the parsers generated by the Yacc and Bison family and passes them to the compiler error message system. It provides the state that the error occurs on and the erroneous token. In conjunction with this information, Merr automatically generates the code of the compiler error reporting function (`yyerror()`) from a set of example errors and messages. 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 [12]. He elaborates on it by designing an algorithm that automatically generates erroneous fragments. Also, he proposed three features of a good erroneous fragment that should be used by the Merr tool: "1) correctness (i.e., every sentence is erroneous), 2) irredundancy (i.e., no two sentences lead to the same parse state), 3) and completeness (i.e., a sentence reaches every parse state where an error can occur)" [12].

III. Analysis of why LR parsers generate incorrect syntax error messages

The parser's role is to take a string of tokens and analyze it to ensure that the string can be generated from the grammar of the source language. If the verification succeeds, parsers build the abstract syntax tree (AST). If the verification fails, the string contains one or more tokens that were not expected according to the given grammar rules. The errors detected by the parser are called syntax errors. Syntax errors can occur for various reasons, such as incorrect token order, missing punctuation, or misspelled keywords. Advanced compilers have error-handling systems that implement one or more error recovery mechanisms. An LR parser reports an error when no valid continuation is related to the current token. LR parsers provide poor error messages when there is insufficient information about the error.

Let us investigate this issue by examining a pair of C++ programs with similar errors as in Fig. 1. In the case of prog1.cpp, the LR parser is able to generate informative error messages. When the parser encounters an issue, it halts at a parsing state, where the expected token is related to recognizing the function body. The parsing states still retain information about the problem. The information available from the LR parser is the state number, the current token, and the expected legal tokens available from the state where the error is detected. In this case, the error handling system can find that the expected token is `'}'` and generate good error messages.

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  }
```

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  }
```

Fig. 1: Listings of a pair of erroneous C++ programs with some common novice syntax errors.

However, in the case of prog2.cpp, the LR parser produces less helpful error messages because it encounters the error in a state where the available information is insufficient to generate a good error message. When the parser reaches `'}'` in line 7, it will recognize the function body and do a group of reduction actions that will leave the parsing stack empty. When it reaches `'cout'` in line 8, it will validate the string of tokens against the rules of global declarations. It will report that `'cout'` is an error even if line 8 syntactically is correct. This will cause incorrect error messages. The parser will suggest expected tokens from the state where the parser halts. These suggestions are misleading to the user, especially novices.

This observation is further supported by the error messages generated by two widely used C++ compilers: GNU GCC version 10.3.1 (GCC) and Microsoft Visual C++ version 2019 (MSVC). Both report the error in prog1.cpp listed in Fig. 1 with good error messages, as in Fig. 2 and Fig. 3. Both report the error in prog2.cpp listed in Fig. 1 with incorrect and misleading error messages as in Fig. 4 and Fig. 5.

```
$g++ prog1.cpp
prog1.cpp: In function 'int recursive(int)':
prog1.cpp:10:2: error: expected '}' at end of input
   10 | }
      |  ^
prog1.cpp:5:22: note: to match this '{'
    5 | int recursive( int x){
      |                      ^
$
```

Fig. 2: Good quality syntax error messages reported by GCC on the compilation of prog1.cpp in Fig. 1.

Error List

Entire Solution | 1 Error | 0 Warnings | 0 Messages | Build + IntelliSense

Code	Description	Project	File	Line	Suppression
C1075	'{': no matching token found	ErrExample	prog1.cpp	5	

Fig. 3: Good quality syntax error messages reported by MSVC upon compilation of prog1.cpp in Fig. 1.

This problem is more expansive than how the parser recognizes the function header and body. It happens when there are errors in the header and body control structures. Fig. 6 lists two C++ programs with common novice errors reported with poor error messages by the LR parser and mainstream

```
$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 | }
      | ^
$
```

Fig. 4: Bad quality syntax error messages reported by GCC on the compilation of prog2.cpp in Fig. 1

Error List

Entire Solution | 11 Errors | 0 Warnings | 0 Messages | Build + IntelliSense

Code	Description	Project	File	Line
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

Fig. 5: Bad quality syntax error messages reported by MSVC on the compilation of prog2.cpp in Fig. 1

compilers. prog3.cpp has a problem with the `if` statements with close and open parentheses. GCC and MSVC reported the error in prog3.cpp with bad error messages as in Fig. 7 and Fig. 8. prog4.cpp is an example of errors in the `for` statements where the user uses commas instead of semicolons, and at the same time, there is an error in the expressions/statements inside the header. In this example, GCC and MSVC cascade a spurious list of errors; instead of reporting simple for statement header punctuation, they report errors in the built-in library and list some code from the library that is not part of the user's source code as shown in Fig. 9 and Fig. 10. This type of error message confuses novice programmers. Section 4 presents a solution to this problem by parsing substrings of source code and validating them to a subset of rules instead of all the grammar rules at once.

IV. New solution: Global, local, expression-level parsing technique

As we discussed in the previous section, LR parsers have difficulties generating good error messages when the available information is insufficient. The parser sees an error. It doesn't know whether it is missing token(s), extra token(s), or token(s) that need to be changed. Worse yet, it does not know the programmer's intentions, and with novices especially, those intentions may be missing or nonsensical. However, LR parsers can be "existentialist" in the sense that they have to give

prog3.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   }
```

prog4.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   }
```

Fig. 6: Additional listings of erroneous C++ programs with common novice syntax errors.

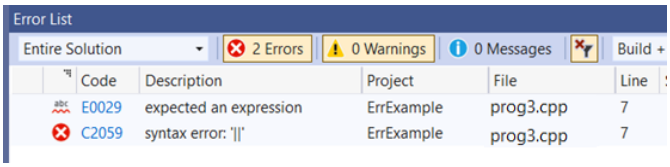

Fig. 7: Bad quality syntax error messages reported by MSVC on the compilation of prog3.cpp in Fig. 6

messages based on what is in the input source code. We attack the problem with the following strategy:

- Divide and Conquer: simplify the parser by focusing on the subset of the production rules that govern the validation of the skeleton of the large component of the input source code. Separating the rules for the outer skeleton from those for the inner skeleton provides advantages in customizing error messages. The number of states in a giant grammar like the C++ grammar is in the thousands.
- The proposed solution uses the lexical analysis to customize the error messages and ignore some erroneous input.
- Implement the topographic debugging strategy, where the programmer should have knowledge about the program

```
$g++ prog3.cpp
prog3.cpp: In function 'int main()':
prog3.cpp:7:15: error: expected primary-expression before '||' token
    7 |   if(a*b<=200)||(b<200)
      |               ^~
$
```

Fig. 8: Bad quality syntax error messages reported by GCC on the compilation of prog3.cpp in Fig. 6

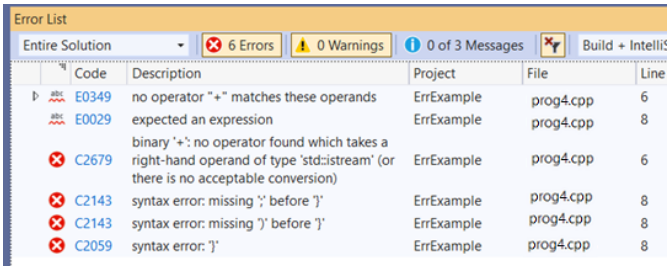

Fig. 9: Bad quality syntax error messages reported by MSVC on the compilation of prog4.cpp in Fig. 6

```
$g++ prog4.cpp
prog4.cpp: In function 'int main()':
prog4.cpp:6:19: 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>}
prog4.cpp:6:19: note: candidate: 'operator+(int, int)' (built-in)
    6 |   for(x=1, y<10, x+
      |                   ~^
    7 |         cin>>y;
      |         ~~~
prog4.cpp:6:19: note:   no known conversion for argument 2 from 'std::istream'
{aka 'std::basic_istream<char>'} to 'int'
In file included from /usr/include/c++/12/string:47,
                 from /usr/include/c++/12/bits/locale_classes.h:40,
                 from /usr/include/c++/12/bits/ios_base.h:41,
                 from /usr/include/c++/12/ios:42,
                 from /usr/include/c++/12/ostream:38,
                 from /usr/include/c++/12/iostream:39,
                 from prog4.cpp:1:
/usr/include/c++/12/bits/stl_iterator.h:630:5: note: candidate: 'template<class
_Iterator> constexpr std::reverse_iterator<_Iterator> std::operator+(typename
reverse_iterator<_Iterator>::difference_type, const reverse_iterator<_Iterator>&)'
  630 |     operator+(typename reverse_iterator<_Iterator>::difference_type __n,
      |     ^~~~~~~~
/usr/include/c++/12/bits/stl_iterator.h:630:5: note:   template argument
deduction/substitution failed:
prog4.cpp:7:9: note:   'std::istream' {aka 'std::basic_istream<char>'} is not
derived from 'const std::reverse_iterator<_Iterator>'
    7 |         cin>>y;
      |         ^~~
```

Fig. 10: Bad quality syntax error messages reported by GCC on the compilation of prog4.cpp in Fig. 6

layout to localize or isolate errors [13].

As a result, the proposed technique parses 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. Merr [11] is used 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 their contents are not parsed at this phase. 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. The lexical analysis of this phase recognizes only the tokens related to the `if` statement, `for` statement, `while` statement, and `switch` structures and validates the rules related them. 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_c()`, and generates an abstract syntax tree (AST-c).

V. Methodology

This study evaluated the innovative model of GLE parsing techniques with an experimental approach. The experiment aims to evaluate the quality of syntax error messages using GLE parsing techniques. It used EduCC, an Educationally Customized Compiler, to generate the error messages. EduCC is a compiler prototype for the C++ language developed for this study that implements the GLE parsing techniques. The experiment compared the error messages of EduCC with the error messages of two 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 with a 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 the mean time to find and fix erroneous programs. Each participant was required to find and fix errors in two programs. One program was accompanied by error messages from EduCC, and another program was accompanied by error messages from MSVC or GCC. For the experiment, we used nine programs sampled from the programs written by students in the introductory programming course seeded with common novice 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. Also, the same program was used once with GCC or MSVC error messages and another with the EduCC error messages. The order of programs and compilers was randomized using Qualtrics. 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 participants were enrolled in lower- and upper-division computer science courses at the New Mexico Institute of Mining and Technology in Fall 2022. Invitations were sent to all the students in the Computer Science and Engineering department. Sixty-six participants responded to the invitation. After cleaning collected data from empty records, the number of respondents became 51. Participants generally needed 25-30 minutes to complete the experiment, but no time constraint was forced. The researchers got the institutional review board (IRB) exemption from the New Mexico Institute of Mining and Technology University for the human-subject experiment.

The experiment was designed as an online self-administered questionnaire using Qualtrics. It consisted of four pages. The first page contained the consent form. On the second page, participants were asked about their age group, gender, and programming experience regarding person-months of coursework or professional experience. The third page presented a program with accompanying error messages, with the compiler's name hidden in the presented messages. Additionally, the third 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 can you fix the error?

Like the third page, the fourth page contained different programs and accompanying error messages generated by different compilers. Pages three and four each had a timer to record the time it took for participants to find and fix the program.

VI. Results

The participants were undergraduate Computer and Engineering Department students, and 90% were 18-24 years old. Though the department has transferred community college students with some experience in programming, 60% of participants have little experience. The percentage of female and male participants is similar to that of students in the department. Tables I, II, and III show the participant percentage over age groups, gender, and programming experience, respectively.

TABLE I: Participants' Age Group

Age group	Percent
18 - 24	90%
25 - 34	8%
35 - 44	2%

TABLE II: Participants' Gender

Gender	Percent
Woman	16%
Man	76%
Non-binary	4%
Prefer not to say	4%

RQ1: Is there a significant difference between the quality of syntax error messages generated by EduCC, and GCC/MSVC in finding syntax errors?
To answer the first research question, three questions from the questionnaire were used as a proxy for the quality of error messages in finding syntax errors. These questions asked: (1) What is the error in the program? (2) In which line is the error located? (3) What is the cause of the error? Two teaching assistants graded the participants' answers, and the compilers' names were hidden. The grading system was as follows: Grade 2 for the correct answer, Grade 1 for a partially correct answer, and Grade 0 for an incorrect answer. The scores for the three questions for each compiler group were totaled, resulting in a maximum score of 6.

A paired samples t-test was conducted to compare the quality of error messages to find syntax errors in two groups. Group One used accompanying messages from GCC or MSVC to answer the questions, while Group Two used accompanying messages from EduCC. The paired-sample t-test results indicate a significant difference between the mean of participants' answers in the two groups (t= 2.6, df=50, p = 0.006). Additional details about the test can be found in Tables IV and V.

RQ2: Is there a significant difference between the quality of syntax error messages generated by EduCC, GCC, and MSVC in fixing syntax errors?
The second research question was addressed using one question from the questionnaire: "How to fix the error?" as a proxy for the quality of error messages in fixing syntax errors. The same two teaching assistants graded the participants' responses, and the compilers' names were hidden. The grading system was as follows: Grade 2 for the correct answer, Grade 1 for a partially correct answer, and Grade 0 for an incorrect answer.

TABLE III: Participants' Programming Expereince

Person-months of coursework or professional experience	C/C++/Python/Java
0-6 months	27%
7-12 months	29%
13-24 months	20%
25-36 months	12%
>36 months	12%

A paired samples t-test was conducted to compare the quality of error messages for fixing syntax errors in two groups. Group One used accompanying messages from GCC or MSVC to answer the questions, while Group Two used accompanying messages from EduCC. The paired-sample t-test results indicate a significant difference between the mean of participants' answers in the two groups (t= 2.5, df=50, p = 0.009). Additional details about the test can be found in Table IV and V.

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?
In order to answer the third research question, a Qualtrics timer was used for each program, which calculates the time the participant spent on the page until the last click. A paired-sample t-test indicates that there is no significant difference in the task completion time between the time used to find and fix errors when the participants used accompanying error messages of EduCC and the time used when they used accompanying error messages of GCC or MSVC (t = -1.70, while the critical t-value is -1.68, df = 50, p = 0.048). We did not limit the time the participants used to find and fix programs. It was observed that some participants spent an unreasonable amount of time on the pages, such as 25 minutes in one program, so the results may not be accurate. Additional details about the test can be found in Table IV and V.

TABLE IV: Paired Samples Statistics

Pair 1	Compiler	Mean	N	Std. Deviation	Std. Error Mean
Finding syntax errors	GCC/MSVC_g1	3.745	51	2.261	0.317
	EduCC_g2	4.529	51	1.419	0.199
Fixing syntax errors	GCC/MSVC_g1	1.367	51	0.929	0.130
	EduCC_g2	1.714	51	0.577	0.081
Time-to-find and -fix	GCC/MSVC_g1	376.215	51	483.984	67.771
	EduCC_g2	273.373	51	322.196	45.116

TABLE V: Paired Differences

Pair1 g2-g1	Mean	Std. Dev.	Std. Err.	95% Conf. Upper	95% Conf. Lower	t-value	df	Sig.(1-tail)	p-value
Finding syntax errors	0.784	2.138	0.299	1.569	0.000	2.619	50	0.05	0.006
Fixing syntax errors	0.333	0.973	0.136	0.667	0.000	2.447	50	0.05	0.009
Time-to-find and -fix	-102.842	431.826	60.468	58.767	-62.169	-1.701	50	0.05	0.048

VII. 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 took a lot of work to recruit students for 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 took a lot of work to separate the time to find errors from the time to fix the error using Qualtrics, and some participants spent

unreasonable time on the page. 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 more help with semantics and run-time errors.

VIII. Discussion

This study is complemented by previous research efforts. GLE parser utilizes the Merr tool developed by Jeffery [11] in the parser to get more information when the parser encounters an error. Part of the compiler community has resorted to handwritten parsers inorder to provide better error messages. GLE and Merr are examples of research that aim to enhance error messages for LR parsers, allowing us to still benefit from automatically generated parsers. Kohn's parser reports error messages for 80% of predefined common errors [3]. However, Kohn reported that the major limitation of their approach is the need to hard-code the predefined errors connected to students' misconceptions. They cannot always write the correct message for the error since they cannot guess the students' intentions from the code. GLE 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 GLE parser is not limited by the program size. Future studies should test GLE parser on large programs. Also, it is expected that GLE parsing technique has the potential to be applied to other programming languages, especially those that are from the C language family, due to the similarities in their grammar.

IX. Conclusion

The research question of this study is: Can the GLE parsing technique help in generating better syntax error messages? This question was evaluated using a controlled experiment and within-group design. The experiments proved that GLE parsing technique enhanced compiler error messages. As a result, they help the user in finding and fixing errors. Using the GLE parsing technique gives compiler writers more control over writing specific error messages for each parsing state and improves the location of reporting problems. The contributions of this study are: 1) analyzing why LR parsers generate bad error messages in some cases, 2) designing an innovative GLE parsing technique, 3) developing a proof-of-concept compiler prototype that demonstrates the utility of the GLE parsing technique for C/C++ languages, 4) measuring the quality of syntax error messages in terms of how well they help users find and fix programs, and 5) conducting an experiment to measure the quality of syntax error messages.

References

[1] J. Hattie, "Bringing it all together," in *Visible learning: A synthesis of over 800 meta-analyses relating to achievement*. Oxford, UK: Taylor Francis Group, 2008, ch. 11, pp. 244–247. [Online]. Available: ProQuest Ebook Central, https://ebookcentral.proquest.com/lib/uidaho/detail.action?docID=367685

[2] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The bluej system and its pedagogy," *Computer Science Education*, vol. 13, no. 4, pp. 249–268, 2003.

[3] T. Kohn, "Teaching python programming to novices: addressing misconceptions and creating a development environment," Ph.D. dissertation, 2017.

[4] S. M. Algaraibeh, T. A. Dousay, and C. L. Jeffery, "Integrated learning development environment for learning and teaching c/c++ language to novice programmers," in *2020 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2020, pp. 1–5.

[5] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin, "Do developers read compiler error messages?" in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 575–585.

[6] P. Denny, A. Luxton-Reilly, and E. Tempero, "All syntax errors are not equal," in *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, 2012, pp. 75–80.

[7] M. C. Jadud, "A first look at novice compilation behaviour using bluej," *Computer Science Education*, vol. 15, no. 1, pp. 25–40, 2005.

[8] J. Jackson, M. Cobb, and C. Carver, "Identifying top java errors for novice programmers," in *Proceedings frontiers in education 35th annual conference*. IEEE, 2005.

[9] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting java programming errors for introductory computer science students," *ACM SIGCSE Bulletin*, vol. 35, no. 1, pp. 153–156, 2003.

[10] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: a case study (at google)," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 724–734.

[11] C. L. Jeffery, "Generating lr syntax error messages from examples," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, no. 5, pp. 631–640, 2003.

[12] F. Pottier, "Reachability and error diagnosis in lr (1) parsers," in *Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 88–98.

[13] C. Li, E. Chan, P. Denny, A. Luxton-Reilly, and E. Tempero, "Towards a framework for teaching debugging," in *Proceedings of the Twenty-First Australasian Computing Education Conference*, 2019, pp. 79–86.