

Evaluating and Improving Syntax Error Diagnostics in Modern Programming Languages

Pinky E. Erandio¹, Reagan B. Ricafort²

School of Graduate Studies, AMA University, Quezon City, Philippines

DOI: <https://doi.org/10.51584/IJRIAS.2026.11080106>

Received: 19 August 2026; Accepted: 24 August 2026; Published: 12 September 2026

ABSTRACT

Syntax errors come up a lot when someone is trying to compile code. The messages a compiler or interpreter shows are one way the system tells the user what went wrong. These warnings and errors are part of how syntax analysis is reported back to the programmer. This work compares how three tools report syntax errors. The tools are Python 3.13.5, Java 21.0.11, and GNU C++ 14.2.0. The goal was to look at the wording and the usefulness of what each one prints. To make the test fair, ten related syntax-error cases were set up. The same kinds of mistakes were used each time. But the source text was adjusted to match each language grammar. The code was not kept identical across all languages. In total, thirty diagnostics were gathered. The data came from running the language implementations directly. No IDE was used. For each diagnostic, six checks were applied. The checks were: clarity, accuracy, where the error was found, specificity, consistency, and usefulness. For the first five checks, a rubric with five levels was used. Clarity, accuracy, location, specificity, and usefulness each got a level from that scale. Consistency was judged in a different way. It was scored at the language level, based on how regular the diagnostic format looked across the ten cases. The results indicate that every tool caught the intended malformed programs. Still, the results were not the same in detail. The kind of info and the way it was written changed across tools. On the six-part scoring scale, Python averaged 4.50. Java averaged 4.63. C++ averaged 4.57. All three tools gave a source location. More variation showed up in specificity. It also showed up in how many alternative messages appeared for one error. These findings describe only what happened under these conditions. They are tied to the exact versions tested. They also depend on the chosen test cases and the scoring method used. Overall, the study adds to work on programming language theory. It links grammar and parsing behavior to the diagnostic layer that reports detected syntax problems.

Keywords: Syntax Errors, Error Diagnostics, Programming Languages, Parsing, Syntax Analysis, Compiler Diagnostics, Error Reporting, Python, Java, C++.

INTRODUCTION

Syntax is the set of rules that tells a programming language how to build valid pieces, like expressions and statements. When a program runs through a compiler or an interpreter, it checks the text against those rules. If the input does not fit any valid pattern, the tool stops and sends an error report to the programmer (Aho et al., 2007; Scott, 2016). So the report is not some side product. It is what you see when lexical or syntax checking fails.

Why the message quality matters is simple. Finding that there is an error is not the same as helping a person fix it. A compiler can reject a bad program for the right reason, yet still give a weak explanation. The location in the code might be off. The message can also be broad, or it may not say what change would help. Work by others shows that many compiler errors are hard to read. It also shows that the way diagnostics are built can change how useful they are, especially for people still learning (Becker et al., 2019; Becker et al., 2021).

This gets more complex across languages. Each language can have its own grammar, its own parsing plan, and its own recovery steps after a failure. Because of that, one real mistake can lead to very different messages. For instance, if a closing delimiter is left out, one system may say the construct is not closed. Another may expect a

token and list what it wanted. A third tool may emit a chain of follow-up errors as it keeps trying to recover (Grune et al., 2012; Parr, 2013). For a fair study or comparison, you have to match the real syntax issue itself, not the exact same text that caused it.

This work looks at a problem in current research: people want to compare syntax diagnostics in a way that can be repeated and checked. The comparison should use clear criteria. An earlier draft suggested doing that, but it did not lock down the exact software versions. It also did not set a single test suite. It did not make the six criteria precise, and it did not show any real diagnostic outputs. The updated study fills these gaps. It runs a side-by-side check of Python, Java, and C++ using the same evaluation setup.

There are two goals. One goal is to test what each implementation reports for the same syntax error cases. The scoring rules are fixed in advance, so results can be compared. The second goal is to note where each system tends to do well or struggle. These notes are meant to point to changes in the diagnostic layer. The goal is not to force changes in the parser or compiler design. In programming language theory, this kind of work touches formal grammar, parsing, how errors are handled, and how language tools are built.

This study stays on syntax diagnostics only. It does not cover semantic errors, runtime faults, speed tests for parsing, or messages that IDE tools may generate. What is analyzed is the information the language system produces after it receives input that is syntactically invalid.

METHODOLOGY

Research Design

The study uses a comparative experimental design. Three language implementations were tested under the same general procedure: prepare a syntax-invalid source file, submit it directly to the language implementation, capture the diagnostic output, and evaluate the output using the same predefined criteria. The experiment was performed in a Debian GNU/Linux 13 (trixie), x86_64 environment. Python files were checked with `py_compile`, Java files were checked with `javac`, and C++ files were checked with `g++` using syntax-only compilation.

Selected Programming Languages and Versions

Language	Implementation / Version	Command Used	Reason for Inclusion
Python	CPython 3.13.5	<code>python3 -m py_compile file.py</code>	Widely used interpreted language with indentation-sensitive syntax and detailed built-in <code>SyntaxError</code> reporting.
Java	OpenJDK 21.0.11	<code>javac Main.java</code>	Widely used statically typed language with explicit block and statement syntax and <code>javac</code> diagnostics.
C++	GNU C++ 14.2.0	<code>g++ -std=c++17 -fsyntax-only file.cpp</code>	Widely used compiled language with a different grammar and diagnostic style.

The evaluation targets the language implementations themselves. No IDE, editor diagnostic, linter, or external static-analysis tool was used. This keeps the collected messages attributable to the selected compiler or interpreter.

Development of Reproducible Syntax-Error Test Cases

Ten test cases were created to represent recurring syntax problems that can be expressed in all three languages. The cases were mapped by underlying grammatical condition rather than by identical source text. This distinction

is necessary because Python, Java, and C++ do not share the same grammar. For example, a missing colon is a meaningful Python syntax error but does not have a direct equivalent in Java or C++; therefore it was not used as a cross-language test case. Likewise, a missing semicolon was not treated as a universal case because Python does not require semicolons for ordinary statements.

ID	Syntax scenario	Cross-language condition
T01	Missing closing parenthesis in expression/call	An opening parenthesis is not closed before the statement ends.
T02	Malformed if condition	The condition begins but its required closing delimiter is missing.
T03	Incomplete expression	A binary operator is left without a right-hand operand.
T04	Unexpected operator sequence	Operators are placed in a sequence that cannot form a valid expression.
T05	Missing separator in initializer	Adjacent elements appear without the required separator.
T06	Unclosed string literal	A string begins but its closing quote is absent.
T07	Unclosed block	A block is opened but the required closing delimiter is absent.
T08	Missing separator in function call	Two arguments/operands appear without a required separator.
T09	Malformed function definition	A function parameter list is incomplete before the function body.
T10	Missing operand inside parenthesized expression	An operator appears before the closing parenthesis without a valid operand.

Equivalent-Error Mapping

For each test case, the source was written idiomatically for the target language while preserving the same underlying syntactic fault. The mapping was performed before collecting diagnostics. Thus, the comparison asks, for example, how each implementation reports an unclosed parenthesized construct, not whether each compiler prints the same words for the same source text. This procedure avoids giving an artificial advantage to a language whose grammar happens to resemble another language more closely.

Diagnostic Evaluation Criteria and Scoring

Each diagnostic was coded using six criteria retained from the original study. To make the criteria reproducible, the first five were scored from 1 to 5 for each test case. Consistency was evaluated once per language because it concerns the regularity of diagnostic presentation across cases. No human subjects or novice-programmer participants were involved; the study evaluates compiler/interpreter output directly. The scoring was performed using the predefined rubric below rather than asking participants to rate messages.

Criterion	1	3	5
Clarity	Only a generic failure is given or wording is difficult to interpret.	The general syntax problem is indicated but requires interpretation.	The wording clearly states the problem in understandable terms.
Accuracy	The message does not correspond to the actual syntactic fault.	The message identifies the general failure but only partially describes the cause.	The message accurately identifies the syntactic problem and its relevant point.

Error Location	No usable source location is provided.	A line or general location is given but provides limited precision.	Line and position/token/caret information clearly narrows the location.
Specificity	Only a generic syntax failure is reported.	An unexpected/expected element or general construct is mentioned.	A concrete missing, unexpected, or unterminated element is identified.
Usefulness	Little practical help is provided for correction.	The message provides some direction but still requires substantial interpretation.	The message gives information that can directly guide correction.
Consistency	Diagnostic format and detail vary substantially for similar cases.	Most cases follow a recognizable pattern with some variation.	Similar cases use a stable location, terminology, and message structure.

Scores of 2 and 4 were used when a diagnostic fell between the descriptions above. For the five case-level criteria, the language score is the arithmetic mean of the ten case scores. For consistency, the language received a single score based on the same rubric. The overall language score is the mean of the six criterion scores. Results were summarized with means and standard deviations for the case-level criteria. Because the sample contains only ten constructed cases per language and the scores are ordinal and researcher-coded, inferential significance testing was not used; the analysis is descriptive.

Diagnostic Collection Procedure

For each case, a source file containing one deliberately introduced syntax fault was created for each language. The file was passed directly to the corresponding implementation. The complete diagnostic output was recorded, including line and column information, diagnostic text, and compiler-provided source excerpts or carets. A test was considered detected when the implementation returned a syntax/compilation error rather than successfully accepting the malformed source. All 30 cases were rejected as intended.

Reproducibility

The experiment can be reproduced with the stated versions and commands. The source forms used for the ten scenarios, together with the captured diagnostic messages and scoring table, should be retained with the manuscript's research materials. Because compiler diagnostics can change between releases, the version numbers are part of the experimental specification and should not be omitted from a later replication.

RESULTS

Diagnostic Detection

All 30 deliberately malformed programs were rejected by their target implementations. This produced a 30/30 detection rate for the selected syntax-invalid cases. Detection rate is reported only as a property of this constructed test suite; it is not interpreted as a general measure of compiler correctness.

Observed Diagnostic Outputs

Case	Language	Diagnostic excerpt	C/A/L/S/U
T01	Python	SyntaxError: '(' was never closed	5, 5, 5, 4, 5
T01	Java	error: ')' or ',' expected	4, 5, 5, 4, 5

T01	C++	error: expected ')' before ';' token	5, 5, 5, 5, 5
T02	Python	SyntaxError: invalid syntax	3, 5, 5, 3, 4
T02	Java	error: ')' expected	5, 5, 5, 5, 5
T02	C++	error: expected ')' before '{' token	5, 5, 5, 5, 5
T03	Python	SyntaxError: invalid syntax	3, 4, 5, 3, 4
T03	Java	error: illegal start of expression	5, 5, 5, 5, 5
T03	C++	error: expected primary-expression before ';' token	5, 5, 5, 5, 5
T04	Python	SyntaxError: invalid syntax	3, 4, 5, 3, 4
T04	Java	error: illegal start of expression	4, 4, 5, 3, 4
T04	C++	error: invalid type argument of unary '*' (have 'int')	3, 3, 5, 4, 4
T05	Python	SyntaxError: invalid syntax. Perhaps you forgot a comma?	5, 5, 5, 5, 5
T05	Java	error: '}' expected (plus follow-on parsing error)	3, 5, 5, 3, 4
T05	C++	error: expected '}' before numeric constant (plus follow-on errors)	3, 4, 5, 4, 4
T06	Python	SyntaxError: unterminated string literal	5, 5, 5, 5, 5
T06	Java	error: unclosed string literal	5, 5, 5, 5, 5
T06	C++	warning/error: missing terminating '"' character	5, 5, 5, 5, 5
T07	Python	SyntaxError: '(' was never closed	4, 4, 5, 3, 4
T07	Java	error: reached end of file while parsing	3, 4, 5, 3, 4
T07	C++	error: expected '}' at end of input	5, 5, 5, 5, 5
T08	Python	SyntaxError: invalid syntax. Perhaps you forgot a comma?	5, 5, 5, 5, 5
T08	Java	error: ')' or ',' expected	5, 5, 5, 5, 5
T08	C++	error: expected ';' before numeric constant	4, 4, 5, 3, 4
T09	Python	SyntaxError: '(' was never closed	5, 5, 5, 5, 5
T09	Java	error: ',', ')', or '[' expected	5, 5, 5, 5, 5
T09	C++	error: expected ')' before '{' token	5, 5, 5, 5, 5
T10	Python	SyntaxError: invalid syntax	3, 4, 5, 3, 4
T10	Java	error: illegal start of expression	4, 5, 5, 3, 4
T10	C++	error: expected primary-expression before ')' token	5, 5, 5, 5, 5

C/A/L/S/U = Clarity / Accuracy / Error Location / Specificity / Usefulness. The displayed excerpts are shortened where the compiler produced repeated source-location or follow-on parsing lines; the scoring was based on the complete captured output.

Criterion Scores

Language	Clarity	Accuracy	Location	Specificity	Usefulness	Consistency	Overall Mean
Python	4.00	4.60	5.00	3.90	4.50	5.00	4.50
Java	4.30	4.80	5.00	4.10	4.60	5.00	4.63
C++	4.50	4.60	5.00	4.60	4.70	4.00	4.57

The highest score for all three languages was Error Location (5.00). The implementations consistently supplied line and position information and displayed a source excerpt or caret in the captured diagnostics. Java obtained the highest overall mean (4.63), followed closely by C++ (4.57) and Python (4.50). The differences are small and should not be interpreted as a statistically significant ranking.

Descriptive Variation

Language	Clarity	Accuracy	Location	Specificity	Usefulness
Python	4.00 ± 0.94	4.60 ± 0.52	5.00 ± 0.00	3.90 ± 0.99	4.50 ± 0.53
Java	4.30 ± 0.82	4.80 ± 0.42	5.00 ± 0.00	4.10 ± 0.99	4.60 ± 0.52
C++	4.50 ± 0.85	4.60 ± 0.70	5.00 ± 0.00	4.60 ± 0.70	4.70 ± 0.48

The standard deviations show that location was completely stable under the scoring rule, while clarity and specificity varied more across the ten cases. Python had lower specificity in cases where it returned the general message “invalid syntax,” whereas it was especially informative in cases where its parser could identify a missing or unterminated construct. Java and C++ generally provided expected-token messages, although C++ sometimes produced multiple follow-on errors from one malformed construct.

Representative Comparison

Scenario	Python	Java	C++
Unclosed parenthesis	'(' was never closed	)' or ';' expected	expected ')' before ';' token
Incomplete expression	invalid syntax	illegal start of expression	expected primary-expression before ';' token
Unclosed string	unterminated string literal	unclosed string literal	missing terminating quote + follow-on errors
Missing separator	Perhaps you forgot a comma?	'}' expected + follow-on error	expected '}' / ';' or ';' + follow-on errors

The representative outputs show why a direct comparison of wording alone would be misleading. Python sometimes gives a highly specific explanation, while Java and C++ often express the same problem through an expected-token diagnostic. C++ can provide technically precise parser information but may also generate several follow-on messages when the first malformed construct affects later parsing.

DISCUSSION

Tests show that error messages can change even if the grammar issue is basically the same. In the run with 30 bad programs, every one of the three systems refused them. Still, the wording of the failure note did not match. Each message pointed to the cause in its own way. That outcome backs the main claim of the work. Fixing syntax is one thing, and telling the user what went wrong is another. They connect, but they are not the same step.

For the Error Location part, all three systems scored a perfect average of 5.00 with the current scoring rules. A key benefit here is how the diagnostic output uses a place in the code. It also marks the nearby text in a way you can spot fast. This helps a lot when the parser waits and only then flags the problem after it has read more tokens. In those cases, the line or offset in the report would be harder for a person to figure out on their own (Aho et al., 2007; Grune et al., 2012).

Big gaps showed up under Specificity. In Python, some failures came with clear messages. Examples include “('(' was never closed,” “unterminated string literal,” and “Perhaps you forgot a comma?” Still, for other broken expressions, Python fell back to the vague “invalid syntax.” Java usually gave messages like “')' expected” or “illegal start of expression.” These help with the structure of what the parser expected, but they do not always

say the likely cause in simple terms. C++ often listed what token should have appeared. Even so, once the first mismatch happened, later errors could show up as well.

These outcomes also point to why equivalent error mapping matters. If a separator is missing in a Python list, it often looks like a comma that was left out. In contrast, Java and C++ may treat the same kind of omission through their own initializer rules, so the message can name a different expected token. If you assume the code was the same in every case, you end up mixing grammar details from each language with differences in how good the message is.

Overall, the scores sit near each other. Java is 4.63, C++ is 4.57, and Python is 4.50. So the data does not back a strong idea that one choice is best for all cases. What the results point to is more specific. Diagnostic quality seems to shift with the syntax mistake itself. It also depends on how each system models what the parser expects. In other words, a language can help a lot for one kind of error, yet be less helpful for a different one.

These results lead to several concrete tips for the diagnostic part of the toolchain. First, when the parser has enough context, the message should name the exact syntax fault. Next, report the most helpful spot in the code. A line and a character or position marker works well. After that, when it is reasonable, say what token or syntax was expected or missing. Also, for errors that are related, keep the wording and the layout steady. Finally, when multiple errors show up in sequence, try to separate the main issue from the follow-on ones. That way, a developer will not feel pushed to fix secondary symptoms first. This matches prior findings about compiler output that people can actually act on (Becker et al., 2019; Becker et al., 2021).

The study also frames what “improving diagnostics” means. The suggestions do not ask for a full swap of the grammar, the parser, or the overall compiler design. You can keep the same syntax analysis. At the same time, you can upgrade the diagnostic layer that turns parser state and token expectations into messages for users. That keeps the work in line with the original goal.

Limitations

- The evaluation uses only three language implementations and ten deliberately constructed syntax-error scenarios.
- The scoring is researcher-coded using a predefined rubric and does not measure how novice or experienced programmers actually understand the messages.
- The results are tied to Python 3.13.5, OpenJDK 21.0.11, and GNU C++ 14.2.0; other releases may produce different diagnostics.
- The test suite focuses on syntax errors and excludes lexical, semantic, runtime, and IDE-generated diagnostics.
- The study uses descriptive statistics because the case set is small and the rubric scores are ordinal; the results should not be treated as an inferential ranking of the languages.
- Some C++ diagnostics include follow-on errors, making it difficult to isolate a single message from the parser's broader recovery behavior.

Implications for Theory of Programming Languages

The study demonstrates a practical connection between formal language concepts and compiler behavior. A grammar defines the structures that valid programs must follow; parsing attempts to recognize those structures; and diagnostic generation communicates where the recognition process failed. Comparing the resulting messages makes visible how grammar, parser expectations, error recovery, and diagnostic design interact. This makes

syntax diagnostics a useful applied topic within Theory of Programming Languages rather than merely a usability feature of development tools.

CONCLUSION

We tested syntax error messages from Python 3.13.5, Java 21.0.11, and GNU C++ 14.2.0. We used ten scenarios that mix languages. Each scenario had a syntax break on purpose. Across the runs, we collected 30 diagnostics. Every bad program was rejected, which matched what we expected. We also updated the method from the first conceptual draft. The main changes were practical. We fixed the code, set the exact versions, and wrote test inputs that could be run again. We described how errors line up across languages. We turned six evaluation points into something measurable. We then used the same scoring steps for all three tools.

The outcomes look similar at first. All three tools point to error spots well. After that, the gaps show up. The differences are clearer in how specific the message is. They also show up in how much extra detail appears after the first broken construct. Python was often very precise for missing pieces and for unterminated parts. Even so, it sometimes returned a plain message like “invalid syntax.” Java often listed tokens it thought should be present. C++ usually gave parser expectations with more technical detail. But it could also produce several chained messages.

I saw average scores of 4.50 for Python, 4.63 for Java, and 4.57 for C++. Since the study is small and meant to describe, treat these numbers as results for the test set used here. Do not read them as a broad ranking of programming languages. The value of this work is mainly the comparison method, plus the check that diagnostic behavior can be studied in a steady way with practical rules.

From the messages we collected, better diagnostics should first focus on what went wrong in the syntax. They should also point to the right spot in the code. It helps to say what was expected or what is missing, but only when we can infer that. Terms should stay consistent across reports. The reports should also separate the first real error from follow on errors. In the next phase, the test set can be larger. More languages and more compiler versions can be added. You can also ask other people to judge the output, or use programmers who are not tied to the authors. It would be good to see if clearer diagnostics actually shorten fix time and reduce confusion. The same approach can be used for lexical, semantic, and runtime diagnostics. Work can also test parser error recovery methods.

REFERENCES

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, techniques, and tools* (2nd ed.). Pearson.
2. Becker, B. A., Denny, P., Pettit, R., Bouchard, D., & Luxton-Reilly, A. (2019). Compiler error messages considered unhelpful: The landscape of text-based programming error message research. *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR)*, 177–210. <https://doi.org/10.1145/3344429.3372508>
3. Becker, B. A., Glanville, G., Iwashima, R., McDonnell, C., Goslin, K., & Mooney, A. (2021). Effective compiler error messages for novice programmers: A systematic literature review. *ACM Transactions on Computing Education*, 21(4), Article 28. <https://doi.org/10.1145/3450603>
4. Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J. H., & Langendoen, K. G. (2012). *Modern compiler design* (2nd ed.). Springer.
5. Parr, T. (2013). *The definitive ANTLR 4 reference* (2nd ed.). Pragmatic Bookshelf.
6. Parr, T., & Fisher, K. (2011). LL(*): The foundation of the ANTLR parser generator. *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 425–436. <https://doi.org/10.1145/1993498.1993548>
7. Scott, M. L. (2016). *Programming language pragmatics* (3rd ed.). Morgan Kaufmann.