PereFlex: A tool for automated evaluation of error recovery in parsers

Olga Bachishche
ITMO University
Saint-Petersburg, Russia
bachisheo@yandex.ru

Yaroslav Vorobiev
HSE University
Saint-Petersburg, Russia
yaroslav.vorobev-2015@mail.ru

Grigory Raykin
ITMO University
Saint-Petersburg, Russia
gregra@mail.ru

Daria Vasina
ITMO University
Saint-Petersburg, Russia
dashavasina625@gmail.com

Daniil Shushakov
ITMO University
Saint-Petersburg, Russia
shushakov4@ya.ru

Semyon Grigorev
Saint Petersburg State University
Saint-Petersburg, Russia
s.v.grigoriev@spbu.ru

***Abstract*—Error recovery is a critical component of parsing technology, particularly in applications such as IDE and compilers, where a single syntax error should not prevent further analysis of the input. This paper presents PereFlex — a tool for extensive experimental evaluation of error recovery in JVM-based parsers. Our evaluation is based on a real-world parsers for Java and users erroneous programs. The results demonstrate that while some strategies are fast, they often fail to provide meaningful recovery, whereas advanced methods offer better recovery quality at the cost of increased computational overhead.**

***Index Terms*—error recovery, parsing, IDE, evaluation**

I. Introduction

An integrated development environment (IDE) is a tool designed to assist developers in writing code efficiently. The features and inspections provided by an IDE are largely based on the abstract syntax tree (AST), which serves as a structural representation of the source code generated by a parser or syntax analyzer. During the coding process, programmers do not always maintain syntactically correct code.

To work effectively under such circumstances, IDE parsers must be capable of recovering from errors and generating an AST that is, if not entirely accurate, at least a reasonable approximation of the correct structure. Efficient error recovery ensures that minor syntax errors do not interrupt the development process [1], enabling features such as autocompletion, real-time syntax highlighting, and quick fixes to function smoothly.

Evaluation of error recovery is important from both scientific and practical standpoints. Parser developers need reliable benchmarks and metrics to compare new recovery algorithms against existing solutions. Likewise, practitioners who build tools such as IDEs, linters, or static analyzers must be able to choose a parser that meets their needs. This requires access to measurable, reproducible information about how well a parser handles erroneous input. Without a clear evaluation framework, it is difficult to make informed decisions or track progress in error recovery techniques.

Despite its practical importance, evaluating the quality of error recovery remains a challenging and largely open problem. First, there is no standardized metric for measuring recovery quality. Different approaches employ a variety of techniques, including analysis of error type distributions [2], exact match metrics [3], and manual assessment [4], [5]. Second, not all metrics are universally applicable. Some are suited to specific parsing algorithms [6], while others depend on the structure of the recovered AST [5], [7]. Different parsers often produce different ASTs for the same incorrect input. This variation arises from differences in their grammars, error recovery strategies, and design decisions. As a result, directly comparing parsers becomes a challenging task.

In addition to metric design, dataset construction presents another fundamental challenge. Many prior studies rely on synthetic datasets, fully generated or partially modified by mutators [5], [7], [8]. These datasets make it possible to automatically label error types and run controlled experiments. However, they can create error patterns that do not reflect real-world code, which limits how well the results apply in practice. To address this, recent studies [1], [6], [8] have started using datasets of real code written by developers [9]. These datasets provide more realistic testing conditions and lead to more reliable evaluations.

This work addresses these challenges by introducing a unified way to evaluate parser error recovery. The proposed method does not depend on a specific parsing technique or AST representation and can be applied to a wide range of real-world codebases.

To ensure an efficient and systematic evaluation, we first review existing approaches to assessing error recovery quality. Building upon this foundation, we present a novel tool that automatically evaluates the error recovery quality of parsers targeting the JVM platform. Notably, the tool can operate without access to a parser's internal structure, including its parsing algorithm or specific error representations.

Additionally, we analyze the error recovery capabilities of widely used JVM-based parsers integrated into IDEs such as VS Code and Eclipse. To facilitate benchmarking, we have annotated a real-user dataset [9] of Java source files containing

syntactic errors.

This research contributes to the field of parsing and syntax analysis in the following ways.

- **Evaluation Metrics**: We review existing approaches to error recovery evaluation and summarize their advantages and limitations.
- **Tool Implementation**: We present **PereFlex**[1] (**Flex**ible **P**arser **E**rror **R**ecovery **E**valuation), a tool that computes a subset of these metrics for any given JVM-based parser without requiring internal access to its implementation.
- **Analysis of Real-World Parsers**: We apply PereFlex to evaluate both generated and standalone parsers used in real-world development environments, including Visual Studio Code[2] and Eclipse[3], providing insights into their recovery quality and limitations.

II. Evaluation Methodology

Many techniques exist for evaluating error recovery [10]. We limit our scope to parser recovery in general, excluding lexer-level errors.

A. Quality Metrics

The central question in quality assessment is how closely the recovered input matches the original user input or a correct reference version. Several evaluation strategies are widely used.

1) The simplest approach involves **manual evaluation** [4], [5]. However, this method is time-consuming and prone to subjective bias. Additionally, developers may be influenced by error-messages output from the tools used during verification [6]. Despite its limitations, manual inspection remains valuable method, particularly in cases where no baseline file with corrected errors is available. To enhance precision, some studies classify recovery quality using coarse categories such as “excellent”, “good”, and “poor” [4]. However, this classification lacks granularity and does not allow detailed analysis.
2) **AST comparison** assesses structural similarity between recovered and expected syntax trees [5], [7]. This method is accurate but sensitive to AST format differences and parser-specific representations. This method also requires a dataset with reference recovered code to compare trees in same format.
3) **Edit distance** [11] quantifies the number of modifications needed to transform one string into another. In error recovery algorithms, particularly in syntax analysis, edit distance is employed to identify the optimal recovery path with minimal data loss [6]. While useful, it may not fully reflect structural differences in complex codebases, requiring careful application.
4) **Exact match** [12] is 1 if the recovered code exactly matches the marked target code, and 0 otherwise. Because it is based on a labeled dataset, false positives may occur when performing alternative but syntactically correct fixes.
5) **Cascade errors**, where one error triggers additional parsing failures, are measured by counting total error locations [6]. Yet, fewer errors do not always imply better recovery, as some parsers may miss detecting them entirely.
6) **First-error stopping** [2], [13] limits recovery assessment to the initial error, ignoring parser behavior for subsequent code, which is vital in IDE scenarios.
7) **Error type distribution** [2] reflects the frequency and kinds of errors encountered. Though often mixing compile-time and parsing errors for Java code, it reveals patterns that highlight parser weaknesses.

In our approach, we selected **edit distance** and **error distribution** as our primary evaluation metrics. Edit distance provides a general, language-independent measure of how well the recovered input matches the original one, while error distribution highlights systematic flaws in recovery strategies, providing insight into the parser’s robustness and practical utility.

B. Performance and Memory Usage

The most straightforward way to evaluate recovery performance and memory consumption is to measure the difference between parsing benchmarks for a correct file and the same file containing errors [5], [7]. This approach eliminates common overhead costs but is only applicable to generated datasets where an ideal solution is known. When working with real-world datasets, performance trends must be compared across different parsers rather than against a predefined baseline.

In some cases, performance measurement can be further refined. For instance, if recovery is a distinct module within a parsing algorithm, its execution time can be measured separately [6]. However, this is only applicable to parsers with dedicated recovery modules.

C. JVM-specific aspects of evaluation

In this study, we analyze the performance of the JVM (Java Virtual Machine). Our methodology is based on “Pro.NET Benchmarking” [14], which provides an approach to experiment design, statistical analysis, and bias mitigation that can be applied to any virtual machine.

- **Use release builds with optimizations**: this ensures that benchmarks reflect real-world tool performance.
- **Prevent compiler optimizations from eliminating measurements**: partial results should be stored in variables to ensure correctness.
- **Repeat benchmarks multiple times**: each performance measurement should be executed at least 30 times, with 5–10 warm-up iterations to stabilize results.
- **Trigger garbage collection (GC) between measurements**: since GC can introduce variability, forcing a collection cycle before measurement ensures consistency.

[1]PereFlex source code: https://github.com/dsult/parser-compare
[2]Visual Studio Code code editor: https://code.visualstudio.com/
[3]Eclipse IDE for Java: https://eclipseide.org/

Since this study focuses on evaluating JVM-based parsers, memory measurement must account for JVM-specific constraints. The most critical metric is peak memory usage during parsing. JVM-based programs do not have a fixed point where peak memory usage is guaranteed, as GC can reclaim memory at any time.

One possible solution is to use specialized profiling libraries such as JMH[4]. However, these tools are designed for microbenchmarking and are not well suited for experiments involving large datasets or extensive computations. An alternative approach is to constrain the JVM's maximum memory allocation and use binary search to determine the minimum required memory for processing a given file.

In summary, a tool for reliable parser evaluation on the JVM requires awareness of its runtime behaviors, especially GC and JIT optimizations.

III. Error recovery analysis tool for JVM platform

PereFlex provides an efficient and structured way to benchmark error recovery capabilities across different parsers on the JVM. It operates as a console application, enabling users to evaluate multiple parsers for different programming languages with precise control over execution parameters.

A. System Architecture

The high-level architecture of the benchmarking tool consists of the following key components shown in Figure 1.

- **RecoveryAnalyzer:** An API for evaluating any JVM parser in a benchmark system and its implementation for different parsers.
- **Benchmark Runners:** Kotlin module that performs multiple measurement algorithms with a given implementation of `IRecoveryAnalyzer` and dataset. This module can be extended by users.
- **Data Representations:** Python library processes the collected benchmark data. This module generates graphs for a specific parser or compares graphs for all analyzers and calculates some statistics to obtain test results.

B. Key Features

The benchmarking tool currently implements the following evaluation metrics for error recovery in parsers.

- **Error Types:** Categorization and analysis of errors encountered during parsing.
- **Similarity:** Similarity score is calculated between recovered output and expected result. In current implementation original token sequence is used as the expected result. The system can be extended to use a reference-fixed version, enabling broader dataset evaluation and experiment design flexibility.
- **JVM Heap Memory Usage** Tracks peak JVM-heap memory consumption during parsing.
- **Processing Time:** Execution time is measured multiple times for each benchmarked input. Each measurement is recorded, allowing users to specify the number of warm-up runs and actual executions even after evaluation.

The system is designed to be extensible, allowing users to define and integrate their own evaluation metrics. By following the existing benchmarking framework, users can implement custom analyzers to gather additional insights, such as parser-specific error handling strategies, CPU usage, or other performance indicators. This flexibility makes the tool adaptable for different research needs and evolving parsing technologies.

C. Conclusion

The evaluation tool described in this paper is a practical solution for researchers and developers working on JVM-based parsing. The tool is also scalable, allowing users to add their own JVM-based parsers and define custom statistical measurements, making it adaptable for different research needs and parser implementations.

IV. Experiment Design

This section describes our approach to measuring the quality of error recovery in parsers.

A. Dataset

To evaluate the quality of error recovery, we use the real-world dataset from Blackbox, a repository of events from the BlueJ Java IDE [15]. This dataset has previously been used for experiments in the field of syntactic analysis [1], [6], [8], enabling direct comparisons with prior research.

B. Research Questions

We design experiments to address the following research questions.

RQ1: How does parsing implementation affect error detection quality?

RQ2: Does similarity score reflect the quality of error recovery?

C. Evaluation Metrics

In this study, we adopt **edit distance** and **error type distribution** as the principal evaluation metrics.

The **edit distance** (`ED`) is zero for identical code; however, in our approach, we invert this metric so that the similarity score equals zero if the token sequences are completely different (1).

$$1 - \left(\frac{\mathrm{ED}(\mathrm{original}, \mathrm{recovered})}{\max(|\mathrm{original}|, |\mathrm{recovered}|)} \right) \tag{1}$$

The `original` token sequence is obtained from the lexer phase, while the `recovered` token sequence is collected from parsing results, including error tokens if error nodes appear.

To evaluate the quality of error recovery by **error type distribution**, the dataset must be annotated with error types

[4]JMH source code: https://github.com/openjdk/jmh

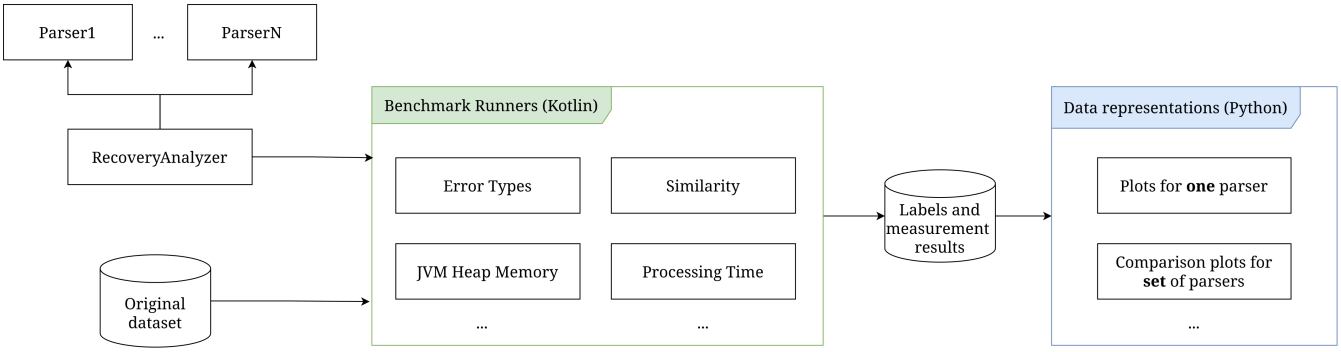

Fig. 1. High-level architecture of the benchmarking tool.

that should be detected. Since the target language is Java, we use the `javac`[5] compiler as a reference. `javac` is part of the Java Development Kit (JDK) so it serves as the de facto standard for compiling Java source files. It provides a well-defined set of diagnostics[6] with human-readable error descriptions[7].

A key challenge is that `javac` does not strictly separate parsing and compilation errors. For instance, a parser error like `illegal start of expression` and a compilation error such as `classes: {0} and {1} have the same binary name` are defined in the same file and referenced by property names. To ensure accurate evaluation of parsing errors, we manually exclude errors from later compilation stages. As `javac` is our reference, the metric for error recovery is the number of errors detected by `javac` that were missed by other parsers.

D. Implementation details

All parser metric evaluations are abstracted using the `IRecoveryAnalyzer` interface, which provides the following methods.

- `getLexerTokens` — returns the set of original tokens obtained after lexing.
- `getParserTokens` — computes the set of tokens after error recovery.
- `measureParseTime` — performs parsing once and returns the execution time.

To support a custom JVM-based parser, it is necessary to implement these methods. During this research, we developed implementations for commonly used parsers and parser generators.

- **Tree-sitter**[8] — A widely used parser generator, notably in VSCode. Originally written in C, it includes optional JVM bundling[9]. Due to its non-heap memory usage, JVM-based memory profiling is not representative, but parsing speed and quality evaluations remain applicable. Since it lacks a built-in lexer API, we implemented an external lexer (based on JFlex for Java) for similarity distance calculation.
- Two versions of **ANTLR** [16] were generated for Java:
  - **ANTLR-Java8-spec** — Based on the official Java 8 specification[10].
  - **ANTLR-Java** — Based on an optimized Java grammar[11].
- **JDT** (Java Development Tools) — Used in Eclipse, JDT differs from conventional parsers by storing keyword tokens (e.g., `if`, `else`, `catch`, `public`) as AST node fields instead of leaves. Extracting token sequences for comparison requires a custom AST traversal mechanism. Fortunately, JDT provides the `NaiveASTFlattener` visitor, which traverses the AST, applies error recovery strategies, and reconstructs the correct code.

In approbation purposes, we analyzed the parsing speed for each parser on the BlackBox dataset. The evaluation results are shown in Figure 2. This demonstrates that our tool enables researchers to compare different parsers in a benchmarking ecosystem with minimal overheads.

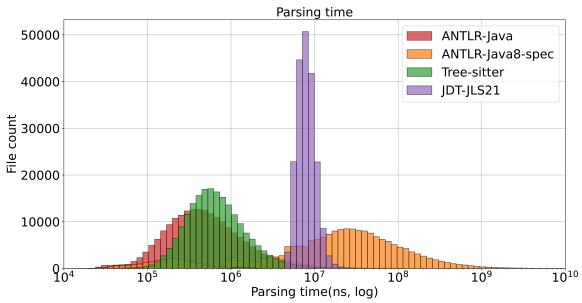

Fig. 2. Parsing time measurement on the BlackBox dataset.

[5]javac: https://docs.oracle.com/javase/8/docs/technotes/guides/javac/

[6]javac diagnostics: https://openjdk.org/groups/compiler/doc/hhgtjavac/diagnostics.html

[7]javac errors: https://github.com/openjdk/jdk/blob/master/src/jdk.compiler/share/classes/com/sun/tools/javac/resources/compiler.properties

[8]Tree-sitter parser generator: https://tree-sitter.github.io/tree-sitter/

[9]Java bundle for Tree-sitter: https://github.com/tree-sitter/tree-sitter-java

[10]ANTLR grammar based on Java 8 specification: https://github.com/antlr/grammars-v4/tree/master/java/java

[11]Optimized ANTLR grammar for Java: https://github.com/antlr/grammars-v4/tree/master/java/java8

V. Evaluation

In this section, we present the evaluation results and address our research questions.

RQ1: How does parsing implementation affect error detection quality?

To assess the impact of parsing implementation on error detection quality, we compare different parsers in terms of missed error messages relative to the reference parser from `javac` compiler. Figure 3 presents the distribution of missed errors across multiple parsing implementations.

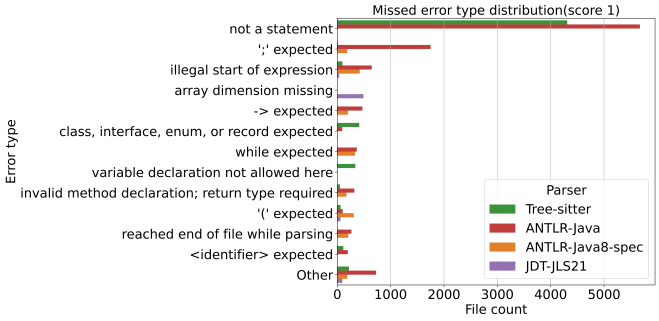

Fig. 3. Missing error distribution in comparing with javac.

The results reveal differences in error detection effectiveness. The most notable finding is the variation in the number of missed errors, particularly for common syntax issues such as `not a statement` and most common Java error [2] `';' expected`. Tree-sitter, for instance, exhibits a high rate of missed `variable declaration not allowed here` and `class, interface, enum, or record expected` errors, indicating its limitations in handling incorrect declarations. ANTLR-Java and ANTLR-Java8-spec, while similar in parsing algorithm, show discrepancies in handling punctuation-related errors such as missing colons and parentheses. JDT, by contrast, appears to have more balanced performance but still misses certain structural errors.

These differences suggest that parsing strategies play a crucial role in error detection quality. Parsers optimized for flexibility, such as Tree-sitter or Antlr-java, may allow faster parsing but at the cost of missing critical syntactic violations. Conversely, stricter parsers such as ANTLR-Java8-spec can enforce the Java grammar more rigorously, but as we can see in Figure 2, they may struggle with parsing performance.

In summary, the choice of parsing implementation directly affects error detection quality. The trade-off between strictness and recovery flexibility determines whether a parser will correctly diagnose syntax issues or silently fail to recognize them.

RQ2: Does similarity score reflect the quality of error recovery?

The similarity score, based on edit distance, measures how closely the recovered code matches the original input. We measured this metric using a BlackBox dataset to evaluate error recovery, as shown in Figure 4. To better understand the quality of the similarity score, we compared its results with other metrics, such as performance and error distribution.

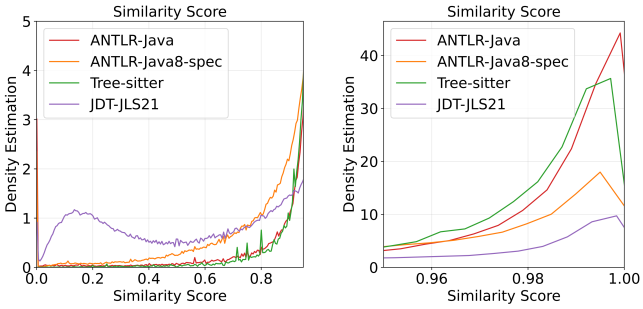

Fig. 4. Similarity score for Blackbox dataset.

Our analysis showed that although the similarity score provides a useful quantification, it does not always reflect the true quality of error recovery.

- **ANTLR-Java** and **Tree-sitter** show high similarity scores, but they tend to miss many syntax errors, as seen in Figure 3. Their outputs look close to the original, yet they often fail to identify and correct issues.
- **JDT-JLS21** has lower similarity scores because it carefully prunes incorrect subtrees during recovery. This leads to more edits but ensures that invalid constructs are not preserved.
- **ANTLR-Java8-spec** closely follows the Java language specification and detects many errors, resulting in more differences from the original and thus lower similarity scores.

In conclusion, while the similarity score alone is not sufficient to assess recovery quality, it still offers valuable insights. It is simple to compute, enables easy comparison between parsers, and it indicates how the recovery process is performed in general. However, since high similarity may also result from ignoring errors rather than fixing them, it must be used alongside correctness and structural validation metrics to accurately evaluate recovery behavior.

Conclusion

In this work, we surveyed and analyzed existing approaches to measuring error recovery quality in parsers. To support subset of this analysis, we developed an open-source tool that enables researchers to evaluate both performance and recovery quality benchmarks for any JVM-based parser. The tool features an extensible architecture, allowing additional benchmarks and data analysis modules to be integrated using a consistent API.

Using this framework, we evaluated a real-world parser on actual erroneous user input. Our results demonstrate that parser recovery quality cannot be fully understood using a single metric alone. Instead, a comprehensive analysis across multiple metrics is necessary to draw meaningful conclusions.

This multifaceted evaluation approach allows not only for a deeper understanding of newly developed parsers but also

facilitates practical comparisons between existing ones. Such comparisons can guide the selection of parsers best suited for real-world applications, depending on whether performance or error recovery is a higher priority for the end user.

VI. Future Work

Impact on Performance: Understanding the impact of error recovery on performance requires proper benchmarking. Establishing a baseline for each example allows us to isolate whether performance variations stem from error recovery or differences in the parsing algorithm.

Cross-Language Evaluation: Extending our framework to other programming languages, particularly those using Parsing Expression Grammars (PEGs) [17], [18], would provide broader insights into recovery strategies and language-specific challenges.

AI-Powered Recovery: AI-assisted tools demonstrate promising results in error recovery by analyzing the AST and suggesting fixes [8], [19], [20]. Future work could explore integrating AI-driven recovery into JVM-based parsers and comparing its quality and effectiveness with traditional methods.

Evaluate more real-world IDE parsers: Some parsers used in real-world projects are not standalone tools but are instead tightly integrated components of larger systems. A key engineering challenge in benchmarking such parsers lies not in supporting them within the benchmarking framework, but in isolating and executing them independently of the full application. Notable examples include the parser from the IntelliJ IDEA IDE and the one embedded in the `javac` Java compiler.

References

[1] I. Karvelas, J. Dillane, and B. A. Becker, "Programmers' views on ide compilation mechanisms," in *Proceedings of the ACM Conference on Global Computing Education Vol 1*, ser. CompEd 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 98–104. [Online]. Available: https://doi.org/10.1145/3576882.3617915

[2] D. Pritchard, "Frequency distribution of error messages," in *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–8. [Online]. Available: https://doi.org/10.1145/2846680.2846681

[3] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large language model for vulnerability detection and repair: Literature review and the road ahead," *ACM Trans. Softw. Eng. Methodol.*, Dec. 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3708522

[4] T. J. Pennello and F. DeRemer, "A forward move algorithm for lr error recovery," in *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '78. New York, NY, USA: Association for Computing Machinery, 1978, p. 241–254. [Online]. Available: https://doi.org/10.1145/512760.512786

[5] M. de Jonge, L. C. L. Kats, E. Visser, and E. Söderberg, "Natural and flexible error recovery for generated modular language environments," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 4, Dec. 2012. [Online]. Available: https://doi.org/10.1145/2400676.2400678

[6] L. Diekmann and L. Tratt, "Don't Panic! Better, Fewer, Syntax Errors for LR Parsers," in *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Hirschfeld and T. Pape, Eds., vol. 166. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, pp. 6:1–6:32. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2020.6

[7] M. de Jonge and E. Visser, "Automated evaluation of syntax error recovery," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 322–325. [Online]. Available: https://doi.org/10.1145/2351676.2351736

[8] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral, "Syntax and sensibility: Using language models to detect and correct syntax errors," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 311–322.

[9] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: a large scale repository of novice programmers' activity," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 223–228. [Online]. Available: https://doi.org/10.1145/2538862.2538924

[10] P. Degano and C. Priami, "Comparison of syntactic error handling in lr parsers," *Softw. Pract. Exper.*, vol. 25, no. 6, p. 657–679, Jun. 1995. [Online]. Available: https://doi.org/10.1002/spe.4380250606

[11] P. Medvedev, "Theoretical analysis of edit distance algorithms," *Commun. ACM*, vol. 66, no. 12, p. 64–71, Nov. 2023. [Online]. Available: https://doi.org/10.1145/3582490

[12] B. Berabi, A. Gronskiy, V. Raychev, G. Sivanrupan, V. Chibotaru, and M. Vechev, "Deepcode ai fix: Fixing security vulnerabilities with large language models," 2024. [Online]. Available: https://arxiv.org/abs/2402.13291

[13] B. A. Becker, C. Murray, T. Tao, C. Song, R. McCartney, and K. Sanders, "Fix the first, ignore the rest: Dealing with multiple compiler error messages," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 634–639. [Online]. Available: https://doi.org/10.1145/3159450.3159453

[14] A. Akinshin, *Pro .NET Benchmarking*. Apress Berkeley, CA, 2019.

[15] I. Utting, N. Brown, M. Kölling, D. McCall, and P. Stevens, "Web-scale data gathering with bluej," in *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ser. ICER '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1–4. [Online]. Available: https://doi.org/10.1145/2361276.2361278

[16] T. Parr and K. Fisher, "Ll(*): the foundation of the antlr parser generator," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 425–436. [Online]. Available: https://doi.org/10.1145/1993498.1993548

[17] S. Queiroz de Medeiros, G. de Azevedo Alvez Junior, and F. Mascarenhas, "Automatic syntax error reporting and recovery in parsing expression grammars," *Sci. Comput. Program.*, vol. 187, no. C, Feb. 2020. [Online]. Available: https://doi.org/10.1016/j.scico.2019.102373

[18] S. Q. de Medeiros and F. Mascarenhas, "Towards automatic error recovery in parsing expression grammars," in *Proceedings of the XXII Brazilian Symposium on Programming Languages*, ser. SBLP '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 3–10. [Online]. Available: https://doi.org/10.1145/3264637.3264638

[19] G. Sakkas, M. Endres, P. J. Guo, W. Weimer, and R. Jhala, "Seq2parse: neurosymbolic parse error repair," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, Oct. 2022. [Online]. Available: https://doi.org/10.1145/3563330

[20] R. Gupta, A. Kanade, and S. Shevade, "Deep reinforcement learning for syntactic error repair in student programs," in *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'19/IAAI'19/EAAI'19. AAAI Press, 2019. [Online]. Available: https://doi.org/10.1609/aaai.v33i01.3301930