A Differential Testing Pipeline for Validating Lambda Expression Handling in Java Compilers Through LLM-Generated Test Cases

Douglas Kosvoski[1], **Andrei Braga**[1], **Rodrigo Ribeiro**[2], **Samuel Feitosa**[1]

[1]Universidade Federal da Fronteira Sul (UFFS)

{douglas.kosvoski,andrei.braga,samuel.feitosa} [at] uffs.edu.br

[2]Universidade Federal de Ouro Preto (UFOP)

rodrigo.ribeiro [at] ufop.edu.br

Abstract. *The Java programming language, known for its robustness and cross-platform capabilities, has evolved with features like Lambda expressions (introduced in Java 8) to enhance developer productivity and code readability. However, the correctness of compiler implementations for such features is critical to maintaining Java's reliability. This study leverages Large Language Models (LLMs) to generate diverse Java programs with Lambda expressions, enabling differential testing of compiler behavior across implementations like Oracle JDK and OpenJDK. Through the proposed approach we were able to validate the consistency of those compilers, and to demonstrate the proposal viability. Additionally, we provide a reusable test-suite framework for evaluating other language features and compilers, demonstrating the utility of AI-driven tools in advancing rigorous software testing and compiler validation.*

Resumo. *A linguagem de programação Java, conhecida por sua robustez e capacidades multiplataforma, evoluiu com funcionalidades como as expressões Lambda (introduzidas no Java 8) para aumentar a produtividade do desenvolvedor e a legibilidade do código. No entanto, a correção das implementações do compilador para tais funcionalidades é crítica para manter a confiabilidade do Java. Este estudo aproveita os Grandes Modelos de Linguagem (LLMs) para gerar diversos programas Java com expressões Lambda, permitindo o teste diferencial do comportamento do compilador em implementações como Oracle JDK e OpenJDK. A partir dessa abordagem foi possível validar a consistência das implementações desses compiladores e demonstrar a viabilidade da proposta. Adicionalmente, fornece uma estrutura de conjunto de testes reutilizável para avaliar outras funcionalidades de linguagem e compiladores, demonstrando a utilidade de ferramentas orientadas por IA no avanço de testes de software rigorosos e na validação de compiladores.*

1. Introduction

The Java programming language has long been valued for its portability, performance, and widespread adoption across industries [Gosling et al. 2000]. Each new release expands its capabilities, incorporating features to address modern programming paradigms. A major milestone was the introduction of Lambda expressions in Java 8 (2014), which brought functional programming concepts into the traditionally object-oriented language [Goetz 2014]. While Lambdas improve code conciseness and readability, they also pose challenges for compiler design and consistent implementation across Java Development Kit (JDK) versions.

Compiler correctness is fundamental to Java's "write once, run anywhere" principle. Inconsistencies between compilers can result in unexpected behavior, performance degradation, or migration failures [Ora 2023b]. Ensuring that Lambda expressions are interpreted uniformly requires rigorous testing against the Java Language Specification (JLS) [Ora 2023a]. Traditional techniques such as manual inspection and unit testing often fall short in detecting subtle bugs, highlighting the need for automated methods that can explore a broader range of scenarios.

Manual test design is limited by human creativity and bias, which can constrain the diversity of generated cases. Automated approaches, by contrast, provide broader coverage and systematically reveal hidden flaws by generating varied and unexpected inputs. As compiler complexity grows, mechanisms that reduce reliance on human effort while improving test thoroughness become essential for ensuring correctness and reliability.

Large language models (LLMs) such as OpenAI's are emerging as powerful tools for automated code generation and testing [Brown et al. 2020]. They can quickly produce diverse test cases that exercise complex features like Lambda expressions, making them particularly useful for differential testing across compilers. This paper leverages LLMs to construct a test suite targeting Lambda expressions, with the goal of uncovering inconsistencies, implementation bugs, or deviations from the JLS in widely used compilers such as Oracle JDK and OpenJDK. The results contribute to assessing compiler reliability while demonstrating the value of AI-driven methods in systematic software testing.

The remainder of this text is organized as follows: Section 2 presents an introduction to Java lambda expressions. Section 3 presents generative code and its use using Open AI's GPT. Section 4 displays works related to this paper. Section 5 delves into the methodology used in this test-suite and evaluation. Section 6 shows the results achieved in this process. Finally, the final remarks are presented in Section 7.

2. Lambda Expressions

Lambda expressions, introduced in Java 8, integrate functional programming concepts into Java's traditionally object-oriented model [Goetz 2014, jls 2024]. They provide a list of parameters and a body—an expression or block—offering a concise syntax for data manipulation and collection operations such as `map`, `filter`, and `reduce` [Oracle Corporation 2023].

```
List<Integer> numbers = Arrays.asList(1,2,3);
numbers.stream()
.map(n -> n * 2)
.forEach(System.out::println);
```

Built on functional interfaces, single-method interfaces that allow lambdas to be treated as objects, this feature reduces reliance on verbose anonymous classes and aligns Java more closely with modern languages like Scala and Kotlin, while maintaining its object-oriented foundation [Gosling et al. 2000]. However, the added complexity also poses challenges for compiler correctness, making consistent implementation of lambda expressions across compilers essential for ensuring code portability [Ora 2023b].

3. Generative Code

Generative code marks a significant advancement in software engineering by automating the production of software components through predefined rules, templates, or machine learning algorithms. This approach reduces repetitive low-level programming tasks, accelerates development, improves consistency, and minimizes human error. Recent progress in artificial intelligence, particularly large language models (LLMs), has strengthened generative programming by enabling contextually accurate and semantically rich code generation. Beyond rapid prototyping, these models also support automated testing and debugging through the generation of diverse test cases, aligning with the principles of scalability and maintainability in modern software development.

Among the most influential advancements are Generative Pre-trained Transformer (GPT) models, built on the Transformer architecture introduced by Vaswani et al. (2017). These models leverage large-scale pre-training and fine-tuning to perform across a wide variety of tasks, including natural language and code generation. The evolution of GPT has been marked by increasing scale and capability, from GPT in 2018 to GPT-4 in 2023, which introduced improvements in reasoning, factual accuracy, and multimodal processing. GPT-4 is notable for its extended context window of up to 128k tokens, allowing it to handle long documents and complex inputs while delivering high-quality outputs across domains ranging from education to software engineering.

For practical integration, OpenAI provides an API that enables developers to embed GPT models into applications without managing the underlying infrastructure. The API abstracts model interaction into simple prompt–response exchanges, offering scalability, security, and ease of deployment. In this work, we employ the GPT-4o Mini model through OpenAI's API. GPT-4o Mini is a lighter, cost-efficient variant of GPT-4 optimized for speed and accessibility, making it particularly suited for automated code generation. Its ability to produce syntactically correct and semantically meaningful programs enables differential compiler testing, significantly streamlining the evaluation process and enhancing software reliability.

4. Related Work

Compiler testing has long relied on random program generation to uncover discrepancies across implementations. McKeeman's pioneering work on differential testing [McKeeman 1995] inspired tools like CSmith [Schkufza et al. 2011], which generates valid random C programs to detect bugs in optimization and execution. Similar approaches have been applied in functional programming, where random well-typed lambda terms have been used to expose subtle issues in Haskell compilers such as GHC [Palka et al. 2011, Kraus et al. 2021, Feitosa et al. 2020].

More recently, the rise of AI-driven methods has expanded compiler testing capabilities. Large Language Models (LLMs) have been investigated both for differential testing of Java compilers and for mutation testing. For example, Wang et al. [Wang et al. 2023] demonstrated that LLMs can automate diverse test case generation, while a subsequent study [Wang et al. 2024] showed that LLM-generated mutants improved fault detection rates by nearly 20% compared to rule-based approaches, albeit with challenges such as higher rates of uncompilable outputs.

Together, these studies illustrate the evolution from random program generators to AI-assisted methods, highlighting the potential of LLMs to complement traditional differential testing by producing richer, more effective test cases for ensuring compiler correctness and software reliability.

5. Test-Suite Implementation

The test suite for this study was designed with two primary components in mind: program generation using OpenAI's API and program compilation targeting multiples compilers. These steps were structured to ensure efficient generation and systematic evaluation of Java programs under test.

5.1. Prompt Engineering

Prompt Engineering is the practice of crafting input prompts to optimize the output generated by AI models like OpenAI's GPT. It involves designing clear, structured, and precise queries or instructions to guide the model toward producing desired results. Prompt engineering is crucial for maximizing the effectiveness of LLMs across various applications, including text generation, summarization, and decision support. Effective prompts can clearly articulate the task or question, while ambiguity can lead to suboptimal or irrelevant responses [Ferrer 2024].

To generate the programs for this study, multiple prompt variations were tested and refined through an iterative process to ensure the output met specific requirements. Each prompt was carefully adjusted based on feedback from test outputs, which helped optimize the generation of clean, functional Java code. The goal was to produce code that was concise, without unnecessary boilerplate such as comments or explanatory text. This refinement process involved several revisions, ensuring that the final prompts reliably elicited the desired responses from GPT-4o-mini. The result was a set of prompts that consistently generated code ready for use, avoiding extraneous content and reducing the need for manual cleaning.

The following and final prompt used in this process specifically instructed the model to generate concise Java programs incorporating lambda expressions, meeting the requirements for this analysis:

```
You are an expert Java programmer.
Your task is to push the limits and explore the boundaries of
   Lambda Expressions in Java, with the intention to break the
   compiler apart to find issues and bugs.
Return only the Java code, with no explanations nor comments.
Include all necessary imports to successfully compile the
   generated code without issues.
Ensure that the main class is named Main and includes a valid
   main method for execution.
Double check everything before outputting the source code.
You can be very creative and explore the boundaries of the
   language to generate the expressions
```

Below is the query that served as the starting point for generating the code to ensure that all relevant parameters and variables were accounted for, allowing for a smooth and efficient code generation process:

```
Generate Lambda Expressions in Java of varying sizes that
   demonstrate various use cases and stretch the limits of the
   feature.
Ensure that the main class is named Main and all the imports are
    included.
```

Deliberate emphasis was placed on guiding the generative AI to prioritize creativity, encouraging it to produce diverse and unconventional Java programs with the potential to uncover compiler bugs and edge cases. The primary goal was to generate programs that could push the boundaries of compiler behavior, deliberately seeking out scenarios that might reveal flaws or limitations in their handling of syntax and semantics.

5.2. Program Generation

To generate the Java programs efficiently, the prompts were set in such a way that the programs could be produced in batches. This was necessary due to rate limiting in the service API, which restricts the number of requests that can be made within a certain timeframe. After generation, each program was stored in the Unix file system, where a unique numeric identifier was assigned as its filename. This approach helped maintain a well-organized system for storing and retrieving generated programs, which was essential for subsequent compilation and testing.

Using unique numeric identifiers, this method ensured that each file could be accessed quickly without the risk of naming conflicts or manual errors. In addition, it facilitated automated processing of the files, enabling batch compilation and analysis to be carried out systematically. This organization is crucial when handling large datasets or numerous program instances, as it simplifies tracking and processing tasks for both quality assurance and testing purposes.

The following pseudocode outlining the steps involved in the program generation phase, which describes the overall process for generating, storing, and organizing Java programs.

```
1. Import necessary modules.
2. Validate API key and set OpenAI API parameters.
3. Set up pagination for response batches
4. Define feature and instructions for generating.
5. For each batch:
    - Send requests for code generation.
    - Parse API response
    - Save each variation to file
6. Log progress and completion status.
```

This pseudocode provides a high-level view of how program generation works from start to finish. It encompasses the initialization of the prompt, the interaction with the service API to generate the code in batches, the saving of the generated programs with unique identifiers, and the subsequent organization of the files in the Unix file system for retrieval and testing. This structured approach ensures that the process is systematic, particularly when working with large quantities of generated code.

5.2.1. Code Generated

The example below is one among over a thousand programs generated during the process described earlier. It illustrates advanced applications of lambda expressions, highlighting their flexibility in handling diverse programming scenarios. Through concise syntax, lambdas simplify complex tasks such as filtering, mapping, and recursion while emphasizing the growing role of functional programming in modern Java. By reducing verbosity and improving readability, they provide more maintainable solutions compared to traditional approaches.

```java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class Main {
  public static void main(String[] args) {
    Consumer<String> varArgPrinter = (argsList) -> {
      Arrays.stream(argsList.split(",")).forEach(System.out::
         println);
    };
    varArgPrinter.accept("arg1,arg2,arg3,arg4,arg5");

    Function<Integer, Function<Integer, Integer>> addLambda = a
       -> b -> a + b;
    System.out.println(addLambda.apply(5).apply(10));

    List<String> names = Arrays.asList("Alice", "Bob", "Charlie"
       );
    Collections.sort(names, (a, b) -> a.length() - b.length());
    names.forEach(System.out::println);

    IntUnaryOperator factorial = new IntUnaryOperator() {
      public int applyAsInt(int x) {
        return (x == 0) ? 1 : x * this.applyAsInt(x - 1);
      }

      public IntUnaryOperator getRecursion() {
        return this;
      }
    }.getRecursion();
    System.out.println(factorial.applyAsInt(5));
  }
}
```

The first part processes a comma-separated string by splitting it into elements and printing them via a stream, demonstrating concise data processing with functional constructs. Next, a higher-order lambda returning another lambda showcases modular composition and arithmetic computation, illustrating the flexibility of functional programming.

In practice, the generated programs share common traits: they range between 20–50 lines and vary in the number and type of lambda expressions used. Each example demonstrates different applications, underscoring the versatility of lambdas in Java. The program shown here was selected randomly, ensuring it is representative without bias toward any particular case.

5.3. Program Compilation

The generated programs were divided into five batches of 250 files each. Compilation was automated with a *Bash* script that handled file management and invoked the target compiler. Each file was copied, renamed to *Main.java*—to match the class declaration and avoid compilation errors—and then compiled. The compiler's exit status was recorded, contributing to success and failure counters for each batch. This procedure was repeated for all files across all batches.

The pseudocode below summarizes the compilation workflow:

```
1. List available compilers
2. Validate the provided compiler argument.
3. Match the argument to a compiler path.
4. Start execution timer
5. Locate all '.java' files in the target directory.
6. For each '.java' file:
   - Copy to a temporary location as 'Main.java'.
   - Compile using the selected compiler.
   - Count successes and failures, log results.
7. Calculate duration and failure percentage.
8. Display compilation summary.
9. Save summary and logs to a file.
```

The process begins by listing available compilers, validating the user's choice, and mapping it to the correct executable. An execution timer is started for performance tracking. The script then iterates through all *.java* files, compiling each after renaming, and logs the outcome.

After all files are processed, the total duration and failure percentage are calculated. A summary of results—successful and failed compilations, execution time, and error rate—is displayed and stored, along with detailed logs containing timestamps and error messages for later analysis.

6. Results

The test suite used JavaScript to generate 1,250 programs across twenty-five batches, with 50 programs produced concurrently in each batch using the OpenAI GPT-4o-mini model.

Experiments ran on an Acer Aspire A515-45 laptop with an AMD Ryzen 7 5700U CPU (4.3 GHz × 8), 16 GB RAM, and Linux 6.8.0-48-generic (Ubuntu 24.04.1 LTS). This setup easily managed concurrent program generation and compilation. Producing one batch of programs required about 30 seconds.

For compilation, the 1,250 programs were divided into five batches of 250. Each batch took about 150 seconds per compiler on newer versions and 90 seconds on older versions.

The graphs below compare results for two Java compilers across older and newer versions. This side-by-side view highlights performance and reliability differences, showing improvements over time.

Figure 1 shows results with Oracle JDK 8.0.202 and OpenJDK 8.0.432, providing baseline metrics for older compilers.

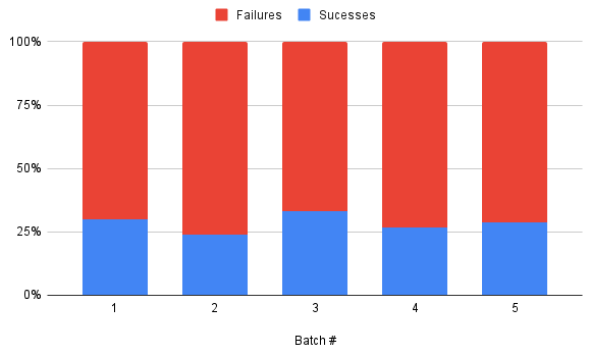

Figure 1. Compilation success and failure counts for each batch on older compilers versions.

Here, success rates remained below 27%, exposing major issues with Lambda expressions in early releases. Shorter compilation times reflected frequent errors, which triggered earlier exits. Both Oracle JDK and OpenJDK showed similar results, confirming the problem was widespread rather than platform-specific.

Figure 2 presents results with Oracle JDK 23.0.1 and OpenJDK 21.0.5, the latest versions at the time. Success rates nearly doubled to around 50%, demonstrating clear progress in reliability and compliance.

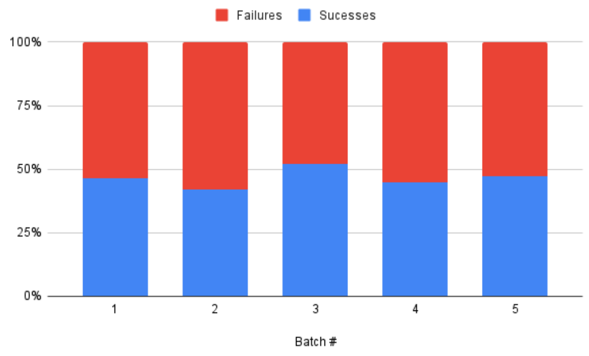

Figure 2. Compilation success and failure counts for each batch on latest compilers versions.

All compilers produced identical results within their version groups, indicating consistency but revealing no new bugs. While this suggests strong reliability, it also shows the test suite was not exhaustive.

Notably, failed code still validated compiler robustness by confirming strict enforcement of Java rules. Many programs failed due to syntax or semantic issues—an expected outcome that nonetheless served as a valuable stress test. This illustrates how generative AI both probes compiler limits and reinforces their resilience.

Overall, AI-driven program generation proved effective for rigorous compiler testing, combining efficient generation, systematic compilation, and wide error exploration into an evaluation framework.

7. Conclusion

In this paper, we described a test-suite framework for differential testing of Java compilers using Large Language Models to generate program codes for a specified programming language feature and verified the compilers' alignment to the Java Language Specification, all the necessary steps were shown, from the prompt provided to the AI model, to the generation and compilation steps. We presented arguments for the importance of generative code to test and stress compilers beyond of what manual and traditional testing is capable of in relation to compilers compliance and cross-compiler support for Java programs, especially for newer features.

As future work, it is possible to expand the test-suite to generate more focused programs by narrowing the programs generated to certain scenarios and improve the compilation step by making use of concurrent programming techniques. Also, the same framework can be explored to test different Java compilers.

References

(2023a). Java language specification (jls). Technical report, Oracle Corporation. `https://docs.oracle.com/javase/specs/`.

(2023b). Java se documentation. Technical report, Oracle Corporation. `https://docs.oracle.com/en/java/`.

(2024). Lambda expressions.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, NeurIPS '20.

Feitosa, S., Ribeiro, R., and Du Bois, A. (2020). A type-directed algorithm to generate random well-typed java 8 programs. *Science of Computer Programming*, 196:102494.

Ferrer, J. (2024). How transformers work: A detailed exploration of transformer architecture.

Goetz, B. (2014). *Java SE 8 for the Really Impatient*. Addison-Wesley.

Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A. (2000). *The Java Language Specification*. Addison-Wesley.

Kraus, L. F., Schafaschek, B., Ribeiro, R. G., and da Silva Feitosa, S. (2021). Synthesis of random real-world java programs from preexisting libraries. In *Proceedings of the 25th Brazilian Symposium on Programming Languages*, SBLP '21, page 108–115, New York, NY, USA. Association for Computing Machinery.

McKeeman, W. M. (1995). Differential testing for software. In *Proceedings of the ACM SIGSOFT '95: Foundations of Software Engineering*, pages 98–104. ACM.

Oracle Corporation (2023). Lambda expressions: The java™ tutorials. `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`. Accessed: 2023-10-28.

Palka, M. H., Claessen, K., Russo, A., and Hughes, J. (2011). Differential testing using random well-typed haskell programs. In *Proceedings of the International Conference on Software Engineering*, pages 504–514. ACM.

Schkufza, E., Sharma, R., and Aiken, A. (2011). Csmith: A tool for generating random c programs. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 320–331. ACM.

Wang, B., Chen, M., Lin, Y., Papadakis, M., and Zhang, J. M. (2024). An exploratory study on using large language models for mutation testing. *CoRR*, abs/2401.04567. OpenReview preprint.

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., and Wang, Q. (2023). Software testing with large language models: Survey, landscape, and vision. *arXiv preprint arXiv:2307.07567*. Available on Papers with Code at `https://paperswithcode.com/paper/software-testing-with-large-language-model`.