Trace-Based Bytecode Interpreter Visualization for Compiler Construction Education

Tobias Herber
Institute for System Software
Johannes Kepler University
Linz, Austria
tobias@herber.space

Markus Weninger
Institute for System Software
Johannes Kepler University
Linz, Austria
markus.weninger@jku.at

***Abstract*—Enhancing students' comprehension of bytecode generation and its interpretation is crucial, yet challenging, in compiler construction courses. Traditional approaches often emphasize theoretical concepts, making it difficult for students to grasp the inner workings of bytecode interpreters.**

To bridge this gap, we introduce a web-based tool for *trace-based bytecode interpreter visualization*, designed to enhance comprehension by providing an interactive and visually enriched learning experience. Our tool provides a side-by-side view that aligns the original high-level source code and its corresponding low-level bytecode, with arrows indicating jumps and method calls. Users can step through the executed bytecode (forwards and backwards) to see the effect of each operation. A dynamic memory visualization utilizes animations to illustrate changes in the interpreter's various memory regions and its registers. To further increase the flexibility of our tool, we developed a lightweight metalanguage that enables educators to define visualizations for arbitrary bytecode formats.

Our tool aims to bridge the gap between abstract theory and concrete execution. We demonstrate its effectiveness in various educational settings, e.g., how it can help educators improve their live teaching and how it facilitates student self-study.

***Index Terms*—Compiler Construction, Bytecode Interpreter, Visualization, Interactive Learning, Debugging, Trace-Based Debugging, Computer Science Education**

I. Introduction

Understanding how a bytecode interpreter works is critical for compiler construction students, yet the dynamic, often opaque, execution process can be difficult to grasp through traditional teaching methods [1]. Visualization techniques are widely recognized for their effectiveness in software engineering education, helping students grasp complex programming concepts through graphical representations and making the process more tangible [2]–[4]. In the context of compiler construction, visualization plays a crucial role in demystifying the complex nature of compiler tooling [1], [5].

Furthermore, visualization has been shown to aid in the development of more accurate mental models, improving students' ability to reason about program execution and compiler behavior [6], [7]. This, in turn, enhances comprehension and engagement, leading to a more effective learning experience.

Graphical debuggers, such as JavaWiz [8], enable users to visualize program execution step-by-step, facilitating a deeper understanding of control flow and the program's inner workings. *Scanner and parser visualizers*, including VCOCO [5] and Jaccie [9], focus on illustrating the inner workings of compilers by visualizing the structured processing of source code. Other tools such as ComVis [1] extend this by presenting different stages of compilation and demonstrating how various compilation approaches influence the overall process.

In contrast to these tools, our tool introduces a novel trace-based visualization approach which allows users to *interactively walk through the execution of bytecode*, even allowing users to step backwards to re-inspect the effect of certain bytecode operations. With each executed operation, the tool animates corresponding changes to the memory state, thereby providing a clear, intuitive window into the inner workings of the interpreter. This step-by-step visualization not only bridges the gap between abstract theory and practical implementation, but also enhances students' comprehension by making invisible processes visible. To the best of our knowledge, no other tool offers this level of detail in tracing memory state transitions during bytecode interpretation, making our contribution a unique asset for both educators and students in the field of compiler construction.

In this paper, we describe the concepts, components, and educational use cases of our visualization tool. In summary, our contributions encompass:

- a trace format combining the source code, bytecode, line mappings, symbol information as well as all executed bytecode operations (section IV),
- a web-based tool visualizing the original high-level source code and the generated lower-level bytecode side-by-side (section V) as well as a highly animated visualization of the interpreter's memory regions and their evolution over time (section VI),
- a set of interactive features for traveling back and forth in bytecode execution, including automatic playback capabilities (section VII), utilizing a DSL for mapping bytecode operations (of various bytecode formats) into high-level animations (subsection VII-D),
- use cases in various educational settings to showcase the applicability and usefulness of our tool (section VIII).

II. Background

Our visualization requires bytecode traces, i.e., a sequential recording of executed bytecode operations. Thus, this section

This is the author's version of the work. The definitive version will be published in the Proceedings of the 13th IEEE Working Conference on Software Visualization (VISSOFT 2025).
A video accompanying this work can be found at https://ssw.jku.at/General/Staff/Weninger/Projects/InterpreterViz/VISSOFT25/Video.mp4.
An interactive demo of the tool can be found at https://ssw.jku.at/General/Staff/Weninger/Projects/InterpreterViz/VISSOFT25/demo.

outlines (1) the general concepts of bytecodes and interpreters, (2) bytecode tracing, as well as (3) the language *MicroJava* (MJ). MJ is used for compiler construction education at the Johannes Kepler University Linz and serves as a reference language for our visualization tool.

A. Bytecode and Interpreters

Bytecode is an intermediate representation of code that consists of compact, lower-level instructions, typically generated by a compiler. Compared to *machine code* (that can only be executed on a given machine type, e.g., `x86`), bytecode enables portability across different platforms. For this, bytecode is executed inside a portable *runtime environment* (such as the *Java Virtual Machine* (JVM)). Such an environment typically utilizes an *interpreter* to execute the bytecode.

A program's bytecode consists of multiple *operations*. Each operation starts with an *opcode* (such as `ADD`, `LOAD`, etc.) and can be followed by *operands* (such as `LOAD 0` to load the first local variable). Within the interpreter, different virtual memory regions are used to manage various types of program data (such as local variables on the stack and global variables in the `data` region). Bytecode operations can be categorized into three main groups:

1) *Read and write operations*, responsible for transferring or copying data between memory regions.
2) *Arithmetic and logic operations*, which perform computations such as addition.
3) *Control flow operations* (e.g., `JUMP` or `CALL`), which alter the execution sequence.

Listing 1 shows a pseudo-bytecode which adds a local and a global variable and stores the result in another local variable.

```
LOAD 1        // Load local var. 1
GETSTATIC 2   // Load global var. 2
ADD           // Add both variables, remember result
STORE 3       // Store result into local var. 3
```

Listing 1: Bytecode example addition.

A crucial focus of our project is to increase students' comprehension by visualizing when and how the interpreter moves data between memory regions (i.e., loads and stores).

B. Bytecode Tracing

In general, an *execution trace* is a recording of a program's executed operations. For our visualization, we collect a trace that contains a list of all bytecode operations executed by the interpreter. Such a trace allows us to inspect a program run in retrospective, for example by employing visualizations that depict the evolution of the virtual machine's state over time.

C. The MicroJava Language and Bytecode

MicroJava [10] is an educational programming language which implements a subset of Java. It has already been used for over two decades in compiler construction education at the Johannes Kepler University Linz. The language and its corresponding MicroJava bytecode include all essential components of major programming languages while maintaining a manageable level of complexity. Although our tool is designed to support arbitrary bytecodes, we chose MicroJava as a suitable initial reference language for our project. We discuss how other languages can be visualized in the future in more detail in subsection VII-D.

III. Approach

This section provides an overview of our visualization's key components. Each component is discussed in greater detail in the following sections.

Our tool is composed of two main subsystems, each serving a distinct role in facilitating bytecode execution visualization:

1) *Execution Trace Collector*: To enable our visualizations, we extend the MicroJava bytecode interpreter and compiler with a lightweight data collection mechanism to collect execution traces in a structured format.
2) *Web-Based Visualization Interface*: Our web interface displays the core visualizations (i.e., a side-by-side view of high-level source code and lower-level bytecode, as well as an animated memory view) and the user interface. It enables users to step through bytecode execution, highlighting important changes on every step.

Figure 1 presents an overview of our system's workflow, illustrating the interaction between its components. The first two components (① and ②) are located in the runtime environment. The *trace collector* (Component 2a) records the execution trace, which is then used by the web application (components ③-⑧) for visualization and user interaction.

① The *compiler* translates source code into bytecode. For our visualization, we record:
- the generated bytecode.
- a bytecode-to-line mapping, enabling us to display generated bytecode next to its original source code.
- symbol information, which establishes a mapping between program entities, such as variables or functions, and their respective source code locations.

② The *bytecode interpreter* receives the bytecode from the *compiler*, executes it, and records a bytecode execution trace for visualization, as explained in subsection II-B.

2a To enable our visualization, the *bytecode interpreter* is instrumented, i.e., adapted, to record execution traces. The *trace collector* collects each executed operation along with the necessary contextual information (for example the results of arithmetic operations) for visualization. Finally, the trace collector serializes (1) the original source code, (2) the generated bytecode, (3) the bytecode-to-line mapping, (4) the symbol information, as well as (5) the execution trace itself into JSON. This serialized data is then provided to the web application.

③ The *code visualization* component processes the program's source code and bytecode to present them next to each other. This helps users better understand the relationship between high-level source code and the corresponding lower-level bytecode [11]. As explained in section V, as the user steps through the execution, the visualization

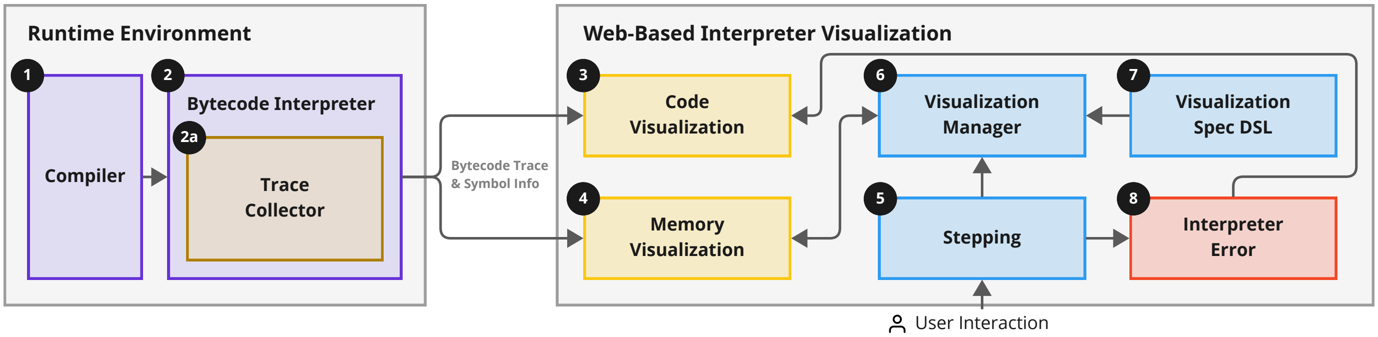


Fig. 1: Overview of our bytecode visualization tool approach. Depicting the interaction workflow of our tool.

highlights the current position in the bytecode, providing an intuitive way to follow the program execution.

④ The *memory visualization* is our tool's core component, providing a representation of the interpreter's memory state. It divides the memory into distinct sections corresponding to the different memory regions of the interpreter, such as the heap for objects and the method stack for locals. Each memory entry, such as a local variable, is displayed within its respective section. Additionally, it incorporates symbol information from Step ①, e.g., it enriches memory entries with variable names and types. The memory visualization and the code visualization (Step ③) work in tandem and are displayed at the same time side-by-side as shown in Figure 2. The memory visualization will be discussed in detail in section VI.

⑤ Our tool enables users to *navigation through the execution trace*, i.e., to revisit all executed operations and their effects on the interpreter's memory. Users can step forwards and backwards one operation at a time using buttons, as well as step multiple operations using a slider. Additionally, users can jump to specific points in the code view, allowing for precise control over the visualization and facilitating in-depth exploration of the bytecode execution flow. We will explain navigation in more detail in section VII.

⑥ The *visualization manager* controls the state of the visualization, primarily the memory view, and is responsible for updating and animating all views accordingly upon user stepping. It derives animations based on a given bytecode operation and the current memory state of the interpreter.

⑦ A core requirement of our approach is to be able to support other bytecodes besides MicroJava. To achieve this, each bytecode operation, along with its behavioral aspects, must be explicitly defined for a given bytecode format. Thus, we have developed a custom *domain-specific language (DSL)* [12], [13]. This DSL allows authors to specify the behavior of bytecode operations, guiding the *visualization manager* (Step ⑥) in interpreting bytecode operations, as detailed in subsection VII-D. By leveraging this mechanism, the visualization can be tailored to different bytecode formats.

⑧ Since our tool is designed primarily for educational use, we cannot assume that the bytecode interpreter or the compiler generating the code will always function correctly. On the contrary, students are even encouraged by our educators to test their in-progress implementations using our visualization tool. To support this, our system is capable of recording interpreter errors, which are then displayed within the visualization to aid in debugging. Additionally, educators can provide reference bytecodes for given source codes, allowing the tool to highlight discrepancies and assist students in identifying issues in bytecodes generated by their own compiler implementations. These issues are displayed in the code visualization and the memory visualization (steps ③ and ④).

IV. Data

To enable our visualization tool, the trace collector gathers a comprehensive set of data beyond just the execution trace. This includes supporting information, such as symbol information, to enhance the clarity and usability of our visualization. The collected data consists of:

1) *the source code, the bytecode, and a bytecode-to-line mapping* – this information is necessary for the code visualization component to place bytecode instructions next to their respective high-level source code line,
2) *symbol information* which is used to associate memory entries with their corresponding variable names and to highlight memory structures such as heap objects,
3) *the bytecode execution trace*, i.e., a detailed log of executed operations, enabling step-by-step playback of the program execution, as laid out in subsection II-B.

All collected data is serialized into a unified format, which can be transmitted to the web application for visualization. This unified format, explained in more detail in the following section, ensures that new trace collectors for different languages can be easily developed. It should be noted that we currently do not consider just-in-time compilation (i.e., code optimizations at run time) as this is out of scope for most introductory compiler construction courses.

A. Execution Trace Format

The execution trace format follows a log-structured approach, maintaining an append-only sequence of data entries. Each entry corresponds to a single executed bytecode operation and consists of (1) *the operation name*, (2) *the operands*,

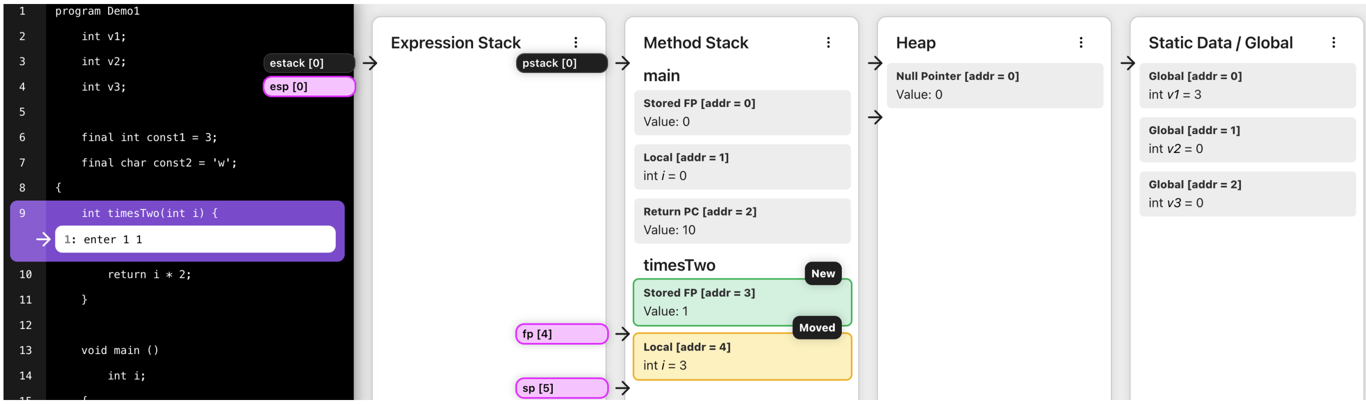


Fig. 2: Our tool's web interface: Code visualization with highlighted current bytecode operation (most-left region) and memory visualization with highlighted updated memory entries and memory pointers (four regions on the right).

i.e., the arguments of the operation, and (3) *operation-specific fields*, capturing all necessary information to apply the operation to the memory state. For instance, in the case of an ADD operation, the corresponding trace entry might look as follows:

```
{ op: "ADD",
  operands: [],
  sources: ["local_4", "static_1"],
  result: 17 }
```

`sources` is an *operation-specific field* which includes information about the input memory entries, while `result` contains the computed result.

Conversely, a LOAD operation for a local variable at address 2 might look as follows:

```
{ op: "LOAD",
  operands: [2] }
```

Additionally, for each executed operation, the execution trace stores updated memory pointers, such as the next available address in the heap or the top of the function stack.

This approach reduces the amount of data collected, especially when compared to approaches that store full memory snapshots. Such snapshots are both computationally expensive and require significant storage [14], [15].

Implementing bytecode tracing required modifications to the MicroJava VM. Specifically, a `recordBytecodeOperation(...)` statement was added to every internal operation handler. Integrating this mechanism into the MicroJava VM was straightforward. Nevertheless, extending it to a more complex VM, such as the JVM, could be more challenging [16]. While our execution trace format is quite minimal and easy to generate, the main challenge is to find every location in the interpreter's source code that executes bytecode operations.

B. Symbol Information

Symbol information is crucial as it enhances the comprehension of memory entries by giving them a meaning. Instead of presenting information in an opaque manner (e.g., "address 5 in the method stack has a value of 42"), we can provide a more meaningful representation (e.g., "variable `foo` of function `bar` has a value of 42").

While numerous sophisticated symbol information formats, such as the Java class File Format [17] and DWARF [18], [19], exist, our system requires only a minimal symbol information format. Our format includes:

- *functions* (name, return type, and start and end position in the bytecode) as well as their *local variables* (address, name and type)
- *static variables*, including their addresses, names and types,
- *representations of structured types*, such as arrays, classes, or structs, specifying their fields and the memory addresses where they are used.

By restricting ourselves to this simple format, we reduce implementation complexity while ensuring compatibility with other, more detailed formats.

C. Error and Anomaly Detection

In addition to recording the execution trace, our format can capture errors encountered by the interpreter. Normal execution is recorded up to the point of failure, providing not only information about the error itself but also enabling the visualization of the bytecode execution leading to the error.

Furthermore, our approach allows for the inclusion of a reference bytecode alongside the actual bytecode. If discrepancies arise between the reference bytecode and the actual bytecode, the differences are highlighted in the visualization, making issues immediately apparent.

This approach aims to foster students' comprehension by providing an intuitive visualization of their own compiler's behavior [3], [4], [6]. It supports the analysis of incomplete or incorrect implementations, helping them identify and resolve issues more effectively. The step-by-step nature of the visualization enables students to investigate execution at their own pace or focusing on specific regions of interest.

V. Code Visualization

The code visualization is an integral component in our tool, complementing the memory visualization, as shown in Figure 2. These two visualizations operate in tandem; while the code visualization highlights the currently executed operation, the memory visualization displays the corresponding memory state at the current point in time. As the user steps through the bytecode execution, the code highlighting dynamically updates to reflect the currently executed operation, providing an intuitive representation of program behavior.

The code section, as displayed in Figure 3a, presents a combined view of both the source code and the corresponding bytecode operations for each line, facilitated by the bytecode-to-line mapping discussed in subsection IV-B.

This combined view highlights the relationship between the source code and the bytecode generated from it, aiding in a better comprehension of this relation. Additionally, it provides a more accessible interface for debugging compilation issues.

A. Operation Highlighting

The code visualization highlights both the current bytecode operation and the corresponding source code line (see colored source code line and bytecode arrow in Figure 3b), updating as the user steps through the execution.

B. Visualization of Control Flow Structures

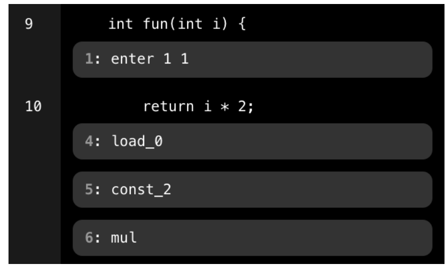
Control flow structures, such as conditional statements and loop statements, are fundamental components of any program. In bytecode, they are implemented using unconditional and conditional jump operations. Even simple conditional statements often involve multiple jump operations, underscoring the need for an intuitive visualization of these mechanisms. Moreover, conditions may be evaluated using *short-circuit evaluation* [20], where condition checks terminate early if the outcome is already determined before all partial conditions are evaluated. For example, in a condition such as `if (5 == 5 || a == b)`, the partial condition `a == b` would never be evaluated due to the first partial condition returning `true`.

To clarify control flow behavior, we employ arrows which indicate the destination of jump operations, as shown in Figure 3c. These arrows are displayed not only for the current bytecode operation or source code line, but also for all lines associated with the corresponding conditional statement or loop statement (e.g., an `if` statement, and all related `else` and `elseif` statements), thereby highlighting all parts of a control flow structure. This way, our approach transforms opaque relative jump addresses into clear indicators, further enhancing comprehension.

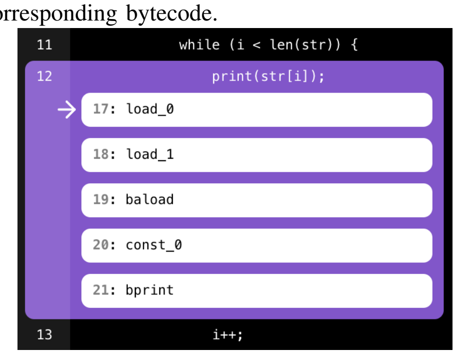
C. Discrepancy Highlighting with Reference Bytecode

As noted in Step ⑧, our tool is designed for use with incomplete or incorrect compiler implementations developed by students. To test their implementations, our students are provided with a comprehensive set of JUnit test cases that check their generated bytecode against a reference solution. Without our tool, mistakes detected by the unit tests are displayed textually as shown in Figure 4a.

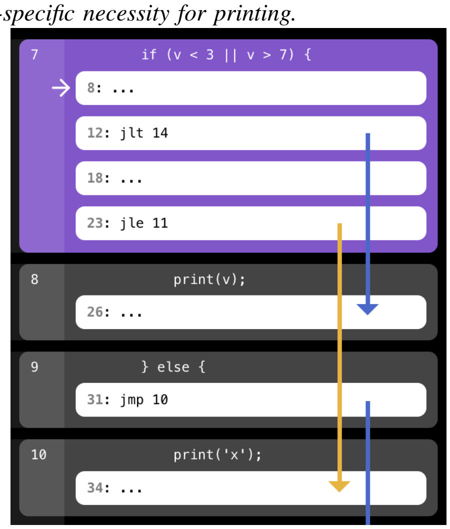
To better help students identify implementation issues, our visualization tool displays the actual generated bytecode alongside the expected bytecode in two columns. Discrepancies, i.e., wrongly generated bytecode, are clearly highlighted,



(a) The code visualization displays both the original source code and the corresponding bytecode.



(b) A highlighted bytecode operation (`load_0`) and its corresponding source code line (`print(str[i])`). *Note: The constant 0 is a MicroJava-specific necessity for printing.*



(c) A simple if statement along with the underlying bytecode. Our visualization includes arrows highlighting jump destinations.

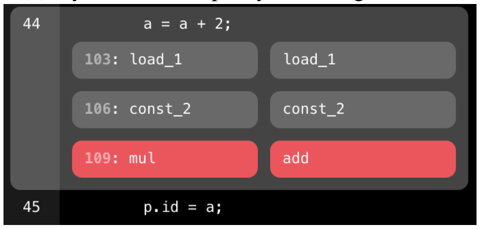
Fig. 3: The code visualization aligns the original source code and the respective generated bytecode.

as shown in Figure 4b. A clear improvement of our visualization over the text-based comparison is based on our bytecode-to-line mapping. This mapping enables students to mentally link broken bytecode ("`mul` is generated instead of `add`") to their source of origin (wrong bytecode generated in line 44 at statement `a = a + 2`)

By offering a more intuitive and engaging alternative to conventional textual outputs, this approach can simplify debugging, making errors visually apparent. Our system facilitates a deeper understanding of the compilation behavior and can aid in more efficient problem resolution [6].

```
  103: load_1           103: load_1
  106: const_2          106: const_2
x 109: mul              109: add
  112: load_1           112: load_1
  115: getstatic_3      115: getstatic_3
```

(a) Bytecode discrepancy in failing unit test.



(b) Bytecode discrepancy in our visualization.

Fig. 4: Textual output from a unit test vs our visualization.

VI. Memory Visualization

This section explores how our tool visualizes memory regions and their entries. Further, we explain how we facilitate animations to increase engagement and comprehension.

A. Visualization of Memory Regions

As outlined in Step ④, the memory of a bytecode interpreter consists of multiple regions (such as heap, stack, etc.), each serving a distinct purpose. A memory region is visually represented as a separate box on the screen. These boxes are positioned next to each other as illustrated in Figure 5. This serves as a clear indicator of which memory entry (e.g., variable or object) is part of which memory region. The memory entries within each memory region are laid out as a vertical list ordered by their addresses.

Additionally, each memory region includes a set of pointers managed by the bytecode interpreter, such as the next-free-address pointer in the heap region. These pointers are visually represented as arrows positioned on the left side of each memory region (for example `data [0]` in Figure 5).

This visualization clearly displays the structure of the memory as well as the individual entries, aiding in a better understanding of memory allocation dynamics.

B. Visualization of Memory Entries

Each memory entry located in a memory region has a set of attributes such as its kind (variable, heap object, etc.), its address or its value. Certain memory entries serve special roles and are explicitly labeled (such as frame pointers or return addresses stored on the stack). Moreover, as described in subsection IV-B, our tool supports displaying symbol information attributes, such as variable names or variable types. This metadata is displayed alongside each memory entry, as illustrated in Figure 6.

Moreover, the memory entry shown in Figure 6 has a dashed outline which indicates that the memory entry is a reference to another memory entry. When the user hovers their mouse cursor over a referencing memory entry, an arrow pointing from the referencing memory entry to the referenced memory entry appears as shown in Figure 7. This can improve the comprehension of pointer-based structures and memory relationships.

C. Animating Changes to Memory Entries

As users step through the bytecode execution, the memory changes. A core objective of our visualization is to clearly illustrate these changes for novices. To do so, our visualization employs various animation techniques for the following operations (see supplementary video):

- *Move and copy operations*: Memory entries are transferred or duplicated between memory regions.
- *Creation operations*: New memory entries are allocated and appended to a memory region.
- *Mutation operations*: Memory entries are modified in place with highlighted changes.

Certain operations involve multiple animation types (for example, function `ENTER` operations where parameters are moved and stack pointers and frame pointers are created). To enhance user control and comprehension, the system allows users to either play all animations automatically in sequence or step through individual sub-animations manually.

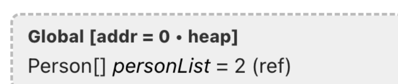


Fig. 6: A memory entry for a global variable `personList` referencing address 2 in the heap (a `Person[]` object). The outlined border indicates that the memory entry is a reference.

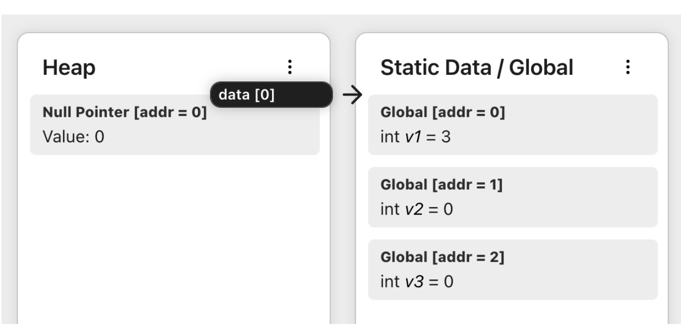


Fig. 5: Two memory region boxes next to each other.

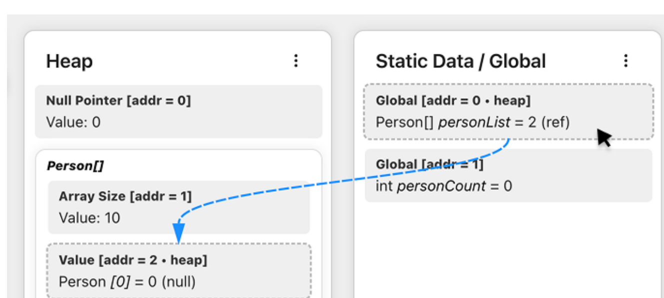


Fig. 7: A hovered reference entry with an arrow pointing to the reference destination.

D. *Visualization of Functions and Structures*

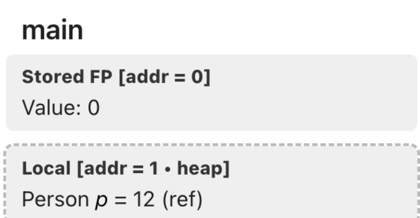
Certain memory entries are grouped together to form structures (e.g., a heap object) or to represent a shared role (e.g., the local variables of a function). For example, memory entries which form the *stack* of a function display the name of the function they comprise, as shown in Figure 8a. Similarly, *structure* memory entries (such as a heap object or struct) are enclosed within a shared box, as depicted in Figure 8b. This box includes a label indicating the type of the structure.

VII. Visualization Management and Bytecode Navigation

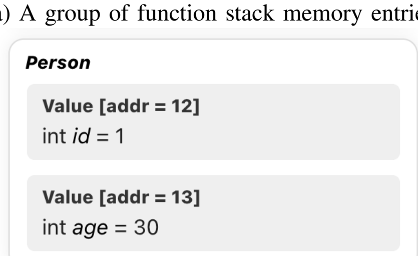
The *visualization manager* manages the state of the entire visualization, including both the code visualization and the memory visualization. As the user navigates through the bytecode execution, the visualization manager updates the memory state, highlights the currently executed bytecode operation, and marks the corresponding source code line. Additionally, since the memory visualization incorporates animations, the visualization manager includes an *animation manager*. This component decomposes memory state changes into smaller, incremental steps, enabling smooth and intuitive animations that enhance the comprehension of execution dynamics.

A. *Stepping through the Execution*

The *animation bar*, shown in Figure 9, serves as the primary navigation interface for stepping through the execution. Unlike conventional debuggers, which typically allow only forward navigation, our approach enables users to move both forward and backward through the execution (a feature known from so-called omniscient debuggers [21]). Additionally, the animation bar includes a slider element which provides a visual indicator of the current execution position. Users can interact with this slider by clicking on any position to jump directly to a specific point in the execution.



(a) A group of function stack memory entries.



(b) A group of memory entries which are part of a structure (in this case a `Person` heap object).

Fig. 8: Memory groups for a function and a structure.

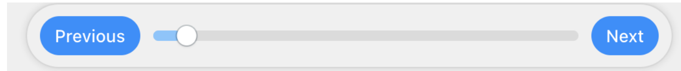


Fig. 9: Navigation bar with forward and backward navigation buttons as well as a slider indicating the current position.

B. *Jumping to Code Positions*

Instead of manually navigating to a specific bytecode by repeatedly stepping, our visualization allows users to directly jump to any position within the execution trace, as illustrated in Figure 10. To achieve this, we traverse the execution trace both forward and backward from the current position, finding the closest trace entry corresponding to the selected bytecode or source code position. Once found, the state of the visualization is updated to the respective execution trace entry. This feature can improve efficiency, particularly for users interested in analyzing specific program segments without manually searching through the execution trace.

Additionally, as shown in Figure 10, our tool provides a *Play until operation* command that performs an *animated* execution playback from the current position until a selected bytecode operation or source code line is reached. Once initiated, every intermediate operation, along with its corresponding animation, is played out sequentially until the target operation is reached. This functionality is restricted to forward execution, meaning users can only play until a future operation. This mode is particularly useful for analyzing how multiple operations work together in a sequence.

C. *Animation Navigation*

As detailed in subsection VI-C, our visualization tool employs animations to intuitively represent memory changes. The *animation manager* decomposes complex memory modifications into multiple sequential animation steps. Using the animation bar shown in Figure 11, users can manually navigate through each step of an animation, providing a clearer understanding of individual memory changes.

For example, for an `ADD` operation, the animation sequence consists of: (1) highlighting the input memory entries, (2)

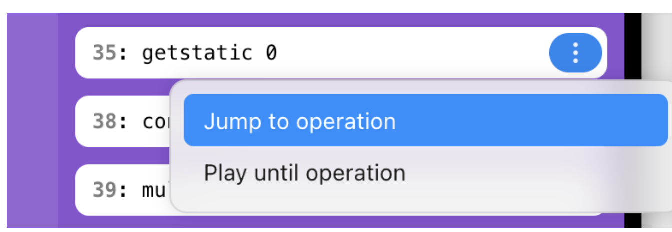


Fig. 10: A dropdown with options to ”Jump to” or ”Play until” the corresponding bytecode operation.

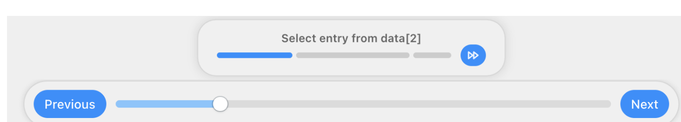


Fig. 11: The animation bar (on top of the navigation bar) with three animations, of which one has already been performed.

creating the addition result, and (3) placing the result in the corresponding memory region. Additionally, users can pause or replay animations as needed with the animation bar, improving control over the visualization process.

D. Visualization Definition DSL

While the execution trace provides a low-level recording of a program's execution, it is not sufficient on its own to generate our memory visualization. Additionally, we want our visualization tool to be able to handle traces for different bytecodes from various bytecode interpreters. To achieve this flexibility, our memory visualization does not have a hard-coded set of bytecode operations it supports. Instead, we introduce a *domain-specific language (DSL)* which enables developers to describe new bytecode formats. These descriptions are then used by our tool to derive visualization updates and animations as the user steps through bytecode.

Firstly, the DSL is used to define the behavior of each bytecode operation. For example, for MicroJava's `ADD`, it defines that the two most-recent loaded values are taken, added, and stored as a new value. Providing such a description for every bytecode operation provides all information necessary for our tool to power its animation engine. The second main use of the DSL is to define the memory regions managed by the bytecode interpreter. For example, this includes the definition of possible memory entry sizes (since different languages provide different kinds of data types in different sizes).

To achieve flexibility across different bytecode formats, our visualization definition DSL employs JavaScript to describe the semantics of each operation. When processing an entry from the execution trace (as laid out in subsection IV-A), the system looks up the definition for the current bytecode operation This DSL-provided specification then guides the visualization manager (outlined in section VII) in applying the necessary state updates and animations.

This approach offers both flexibility and a high-level abstraction for integrating new bytecode interpreters. Moreover, the use of this format allows our visualization and our execution trace format to be unopinionated, thereby enabling support for bytecode interpreter specifics which we may not have thought of. As a reference, the DSL description for MicroJava (which can be found at https://ssw.jku.at/General/Staff/Weninger/Projects/InterpreterViz/VISSOFT25/MJ.vizspec) comprises about 900 lines of code to describe its 5 memory regions (code, heap, stack, global data and expression stack), 8 types of memory entries, and the behavior of its 57 bytecode operations, demonstrating our DSL's conciseness.

VIII. Educational Application Scenarios

Our tool is designed to support a wide range of educational scenarios. In this section, we demonstrate its potential through three representative use cases:

- *Live teaching*, i.e., educators using the tool for live demonstrations in class.
- *Self-directed learning*, i.e., students using the tool at home to inspect the behavior of their compilers and interpreters.
- *Bug fixing*, i.e., students using the tool to find and fix bugs in their implementations.

While we have not yet conducted formal user studies, these examples demonstrate that our visualization can meaningfully support both learning and problem root cause analysis.

A. Live Teaching

Compiler construction courses often rely primarily on slide-based materials to explain abstract execution models and bytecode semantics. However, these materials are inherently limited in their ability to convey the dynamic behavior of bytecode execution and the intricate underlying memory changes that occur within a bytecode interpreter. Our tool addresses this gap by enabling educators to deliver interactive, real-time visual demonstrations of bytecode execution during lectures.

In a typical teaching scenario, an educator can present the visualization tool while walking through the execution of a small example program. As each bytecode operation is executed, the corresponding changes in memory and control flow are animated and clearly shown to students. The direct mapping between source code and bytecode, combined with the animated visualization of the bytecode interpreter's memory states, can foster deeper and more intuitive comprehension of these otherwise opaque and abstract concepts [3], [4]. This visual aid can bridge the gap between theoretical explanations and practical execution, making the material more accessible.

This approach can not only support the comprehension of bytecode interpreter mechanics but also enhance student engagement [6], [7]. It enables educators to walk through complex concepts such as control flow, function calls, and memory operations at the students' pace. Moreover, it encourages active participation by inviting students to interact with the material, facilitates live Q&A sessions, and allows educators to address student questions with concrete, visual examples that clarify doubts immediately. Our *time-travel* capability [8], [21], [22] is particularly useful in such educational contexts, as educators may need to revisit specific steps during live demonstrations upon student request or to explain certain parts in more detail.

B. Self-Directed Learning

Outside the classroom, students often find it difficult to fully internalize the abstract behavior of compilers and bytecode interpreters without direct guidance. Our tool addresses this by supporting self-directed learning [23]–[25], enabling students to load bytecode generated by their own compilers or reference bytecode examples and interactively step through the execution process at a pace that works for their individual needs.

In a typical self-study scenario, a student can explore the mechanics of how a bytecode interpreter operates by interactively stepping through a bytecode execution. Features such as forward and backward stepping, along with jump-to-position navigation, enable students to focus on specific areas of interest, revisit intricate control flow logic, or observe the effect of individual operations on memory states. Moreover, students have the ability to load and examine bytecode produced by their own compilers, thereby gaining deeper insights into how their implementations work or analyzing issues within their implementations. This process provides immediate visual feedback on the correctness as well as the behavior of their implementations, thereby further enhancing comprehension.

Self-directed learning, as facilitated by our tool, can foster deeper understanding and actively encourage experimentation, as supported by prior research [26]–[28]. It enables students to test hypotheses, analyze behavior, and develop a deeper understanding without relying on in-class teaching. Moreover, this approach affords students the flexibility to take as much time as they need to fully grasp complex topics.

C. Debugging and Bug Fixing

Understanding unexpected behavior in bytecode is notoriously difficult, especially for students who are still learning the underlying concepts. Our tool supports interactive debugging, thereby helping users identify the exact moment when the program state diverges from expectations.

In a typical use case, a student encounters a bug, such as an incorrect return value from a function. By stepping through the execution trace, they observe that an erroneous `POP` operation removed a value from the stack, even though it should not have been removed. Our tool's step-back functionality allows them to trace the error's origin precisely. Moreover, if a reference bytecode is provided and discrepancies arise, the bytecode comparison view highlights these discrepancies to assist in identifying faulty compiler output. Additionally, interpreter errors, such as popping from an empty stack, are clearly visualized and annotated. This approach enables a more intuitive form of low-level debugging, offering a structured alternative to ad-hoc print statements. By making internal state transitions and memory changes visible, the tool enables users to reason about bugs with greater confidence.

D. Takeaways for Educational Use

The use cases discussed in this section highlight the versatility and educational value of our visualization. Whether used in live teaching, for self-directed learning, or as a debugging aid, our tool offers valuable capabilities for educational settings. While formal evaluations are pending, these scenarios demonstrate that our visualization tool can enhance students' comprehension of bytecode programs. Our tool does so primarily by (1) allowing for arbitrary navigation through bytecodes, (2) interactively showing the behavior of a bytecode interpreter, (3) establishing a correlation between bytecode and source code, and (4) allowing both students and educators to visually analyze a bytecode execution of their choosing.

IX. Related Work

Our tool positions itself as a valuable addition to compiler construction education. While visualization support in this domain is rather scarce, many other trace-based visualizations exist. Thus, we present various trace-based debuggers, followed by tools for program comprehension, educational visualization tools and finally other compiler-related visualizations.

a) Tracing Debuggers: Traditional debuggers inspect the program during a live execution and can only traverse forward while accessing the current program state [29]–[31]. In contrast, our visualization is inspired by *omniscient debuggers* that allow free navigation through the execution of a program, even backwards through time. The concepts and definitions of *omniscient debuggers* [21], [32]–[36], (online and offline) *time-traveling / reverse debuggers* [8], [30], [31], [37], as well as *capture-and-replay / (record-and-)replay debuggers* [38], [39] often overlap, are mixed, and become blurred. What unites all of these approaches is that they rely on *execution traces* [40]–[44] or traces of system snapshots. Omniscient debuggers and live reverse debuggers typically provide the possibility to step back through time during a live debug session, while offline tools (such as our visualization) rather inspect the recorded trace at a later point in time. Capture-and-replay / (record-and-)replay debuggers typically record application and system state at every program step (including external inputs) and use these to re-execute the application, with the exact same state and inputs, at a later time or even on a different machine.

b) Visualization Tools for Program Comprehension: Numerous systems and research endeavors focus on visualizing various aspects of dynamic program behavior [45]–[48], most often also relying on various kinds of traces. For instance, Weninger et al. [22], [49]–[51] employ diverse visualization techniques, such as memory cities, to provide comprehension support for heap structures, aiding in understanding heap allocations and the identification of memory leaks.

c) Educational Visualization Tools for Programming: Sorva et al. [52] give a very comprehensive tool overview for visualizations in introductory programming education. They highlight that many educational tools focus on specific program aspects or tasks such as visualizing the stack and heap memory structure visualization during program execution. For instance, JAVAVIS [53], a desktop application, offers object and sequence diagrams, pioneering memory visualization for Java. Jeliot [54]–[56] has a strong focus on animations [57] and provides similar functionalities in a graphically more polished fashion. The Netbeans IDE plugin EZVisor [58] visualizes program executions as abstract diagrams. jGRASP [59] is an IDE that features conceptual visualization of data with object viewers. Other examples of object structure visualization tools are Java Visualizer [60], Python Tutor [61], or VIPER [62]. All these tools inspired JavaWiz [8], a modern trace-based time-travel debugger for education. It provides numerous interactive problem-specific visualizations, including automatically generated flow charts, a detailed memory visualization, a tabular view of variable changes, animated visualizations of linked lists and binary trees, as well as 1D and 2D array operations and input buffer monitoring. While all mentioned tools aim to help teaching novice programmers who are (rather) new to programming, they are not very helpful for very specific tasks such as compiler construction education with its need for very specialized visualizations.

d) Compiler-related Visualizations: Visualizing and animating abstract syntax trees (ASTs) can help understand how programs are structured and parsed [63]–[65]. For instance, Debug Visualizer [66] provides, besides many other highly configurable visualizations, features to visualize the AST of a given piece of code. *Scanner and parser visualizers* [5], [9], [67]–[69] illustrate the inner workings of scanners (responsible for *lexical analysis* [70]) and parsers (responsible for *syntax analysis* [10]). Besides scanner and parser visualizations, ComVis [1] also contains a module for teaching *semantic analysis* based on annotated syntax tree visualization. Our work is positioned later in the compilation process, focusing on the *code generation phase* and the *interpretation phase*. While multiple tools exist that visualize bytecode itself [71], [72] (for example as a graph [73]), there is a lack of tools that visualize the actual work of the interpreter, i.e., bytecode execution. For instance, Bytestacks [74] visualizes executed bytecode as a flame graph [75], [76], but it does not provide supporting visualization such as looking at the interpreter's memory as it executes the bytecode. Other kinds of compiler-related visualizations encompass visualizing compiler optimizations [77], [78] or just-in-time compilation visualization [79].

X. Limitations and Future Work

While our visualization tool is fully functional and supports all features outlined in this paper, there are several areas where it could be extended, improved, and evaluated in more detail. This section outlines current limitations and potential directions for future development.

a) User Study: We believe that our visualization is helpful to inspect the execution behavior of bytecode interpreters, especially for novice users who otherwise had no way of looking inside a bytecode interpreter and had to learn exclusively from textbooks or slides. We presented educational scenarios and discussed the potential use of the tool in compiler construction courses. Nevertheless, a systematic evaluation of the tool is currently missing. Thus, we plan to conduct a user study in the upcoming compiler construction courses at the Johannes Kepler University Linz. Our goal is to gain insight into metrics such as *time to find bugs*, *user satisfaction*, and increased *student engagement*, as well as to collect feedback on existing shortcomings and potential future improvements to the tool. Initial feedback in a pilot study already revealed the necessity to enable users to control the amount and speed of animations and to enable them to hide or show certain information based on the context the tool is currently being used in (e.g., classroom vs. self-directed learning).

b) Support Common Bytecode Formats: Although MicroJava has proven effective for prototyping and education, it is not commonly used beyond that. Supporting a more complex and widely adopted platform such as the JVM would not only increase the relevance of our tool but also serve as a valuable test case for our extensibility mechanisms, particularly the bytecode trace format and the visualization DSL. Specific areas of interest are how well our DSL can map more complex memory models (such as Java's type system and heap implementation) and bytecode instructions that are not part of MicroJava (such as Java's `invoke_virtual` or operations related to exception handling).

c) Standardized Symbol Information Format: Currently, our system uses a minimal custom implementation for symbol information. Adding support for a standardized format such as DWARF [18] could improve compatibility and facilitate support for additional programming languages. Such standards are widely used, well documented, and often more comprehensive than custom solutions. Moreover, good DWARF implementations already exist for a wide range of programming languages, enabling us to more easily integrate new bytecode interpreters and their respective programming languages.

XI. Conclusions

Understanding bytecode execution and effectively explaining the abstract concepts of compilers and bytecode interpreters in an engaging manner remains a significant hurdle in compiler construction education. We have introduced a novel trace-based visualization tool designed to improve comprehension by interactively bridging the gap between abstract concepts and concrete runtime behavior. Our visualization allows for interactive stepping (both forward and backward) through bytecode. It presents a unified view of source code and corresponding bytecode, coupled with an animated memory visualization highlighting state changes. Symbol information enhances clarity, and a custom Domain-Specific Language (DSL) ensuring extensibility to various bytecode formats.

This visualization benefits educators by enabling engaging live demonstrations and guides students through self-directed learning, analysis of their own compiler implementations, and more intuitive debugging. By making the intricate details of bytecode interpreters visible and interactive, our tool improves the educational experience in compiler construction courses. It offers a valuable platform for understanding low-level system behavior, making complex processes transparent and accessible, thereby fostering deeper learning and potentially inspiring further advancements in educational visualization tools.

XII. Supplementary Materials

An *artifact*, including the source code, a tool walkthrough, and a video can be found at https://doi.org/10.5281/zenodo.16899998 [80].

References

[1] S. Stamenkovic and N. Jovanovic, "A Web-Based Educational System for Teaching Compilers," *Trans. Learn. Technol.*, vol. 17, pp. 143–156, 2024. [Online]. Available: https://doi.org/10.1109/TLT.2023.3297626

[2] S. M. Cisar, R. Pinter, and D. Radosav, "Effectiveness of Program Visualization in Learning Java: a Case Study with Jeliot 3," *Int. J. Comput. Commun. Control*, vol. 6, no. 4, pp. 668–680, 2011. [Online]. Available: https://doi.org/10.15837/ijccc.2011.4.2094

[3] M. Hu, T. Assadi, and H. Mahroeian, "Teaching Visualization-first for Novices to Understand Programming," in *Int'l. Conf. on Engineering, Technology & Education (TALE)*. , 2021, pp. 654–660. [Online]. Available: https://doi.org/10.1109/TALE52509.2021.9678922

[4] M. Jalalitabar and Y. Wang, "Demystifying the Abstractness: Teaching Programming Concepts with Visualization," in *Conf. on Information Technology Education (SIGITE)*. , 2022, pp. 134–136. [Online]. Available: https://doi.org/10.1145/3537674.3555786

[5] R. D. Resler and D. M. Deaver, "VCOCO: a visualisation tool for teaching compilers," in *SIGCSE Conf. on Innovation and Technology in Computer Science Education (ITiCSE)*. , 1998, pp. 199–202. [Online]. Available: https://doi.org/10.1145/282991.283123
[6] L. Ma, J. D. Ferguson, M. Roper, I. Ross, and M. Wood, "Improving the Mental Models Held by Novice Programmers using Cognitive Conflict and Jeliot Visualisations," in *SIGCSE Conf. on Innovation and Technology in Computer Science Education (ITiCSE)*. , 2009, pp. 166–170. [Online]. Available: https://doi.org/10.1145/1562877.1562931
[7] T. L. Naps, G. Rößling, V. L. Almstrum, W. P. Dann, R. Fleischer, C. D. Hundhausen, A. Korhonen, L. Malmi, M. F. McNally, S. H. Rodger, and J. Á. Velázquez-Iturbide, "Exploring the Role of Visualization and Engagement in Computer Science Education," *SIGCSE Bull.*, vol. 35, no. 2, pp. 131–152, 2003. [Online]. Available: https://doi.org/10.1145/782941.782998
[8] M. Weninger, S. Grünbacher, and H. Prähofer, "JavaWiz: A Trace-Based Graphical Debugger for Software Development Education," in *33rd IEEE/ACM Int'l. Conf. on Program Comprehension (ICPC@ICSE 2025)*. IEEE, 2025, pp. 1–12. [Online]. Available: https://doi.org/10.1109/ICPC66645.2025.00023
[9] N. Krebs and L. Schmitz, "Jaccie: A Java-based Compiler-Compiler for Generating, Visualizing and Debugging Compiler Components," *Sci. Comput. Program.*, vol. 79, pp. 101–115, 2014. [Online]. Available: https://doi.org/10.1016/j.scico.2012.03.001
[10] H. Mössenböck, *Compiler Construction: Fundamentals and Applications*. Springer Nature Switzerland, 2025. [Online]. Available: https://link.springer.com/book/10.1007/978-3-031-84813-1
[11] H. B. Lee and B. G. Zorn, "BIT: A Tool for Instrumenting Java Bytecodes," in *USENIX Symposium on Internet Technologies and Systems (USITS)*. USENIX, 1997. [Online]. Available: http://www.usenix.org/publications/library/proceedings/usits97/lee.html
[12] M. Mernik, J. Heering, and A. M. Sloane, "When and How to Develop Domain-Specific Languages," *Comput. Surv.*, vol. 37, no. 4, pp. 316–344, 2005. [Online]. Available: https://doi.org/10.1145/1118890.1118892
[13] M. Weninger, E. Gander, and H. Mössenböck, "Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection," in *Int'l. Conf. on Performance Engineering (ICPE)*. , 2019, pp. 273–284. [Online]. Available: https://doi.org/10.1145/3297663.3310297
[14] E. Aftandilian, S. Kelley, C. Gramazio, N. P. Ricci, S. L. Su, and S. Z. Guyer, "Heapviz: Interactive Heap Visualization for Program Understanding and Debugging," in *Symposium on Software Visualization*. , 2010, pp. 53–62. [Online]. Available: https://doi.org/10.1145/1879211.1879222
[15] N. Grech, G. Fourtounis, A. Francalanza, and Y. Smaragdakis, "Shooting From the Heap: Ultra-scalable Static Analysis with Heap Snapshots," in *SIGSOFT Int'l. Symposium on Software Testing and Analysis (ISSTA)*. , 2018, pp. 198–208. [Online]. Available: https://doi.org/10.1145/3213846.3213860
[16] J. Maebe, M. Ronsse, and K. Bosschere, "Instrumenting JVMs at the Machine Code Level," in *PA3CT-symposium*, vol. 19, 2003.
[17] Oracle. (2011) The class File Format. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html
[18] DWARF Debugging Information Format Committee. (2025) DWARF Debugging Standard. [Online]. Available: https://dwarfstd.org/
[19] T. Bastian, S. Kell, and F. Z. Nardelli, "Reliable and fast DWARF-based stack unwinding," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 146:1–146:24, 2019. [Online]. Available: https://doi.org/10.1145/3360572
[20] M. H. Clifton, "A Comparison of Space Requirements for Short-circuit and Full Evaluation of Boolean Expressions," in *Southeast Regional Conf.* , 1998, pp. 261–267. [Online]. Available: https://doi.org/10.1145/275295.275367
[21] G. Pothier and É. Tanter, "Back to the Future: Omniscient Debugging," *Softw.* , vol. 26, no. 6, pp. 78–85, 2009. [Online]. Available: https://doi.org/10.1109/MS.2009.169
[22] M. Weninger, L. Makor, and H. Mössenböck, "Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations," in *Smart Tools and Apps in Computer Graphics (STAG)*, 2020, pp. 63–75. [Online]. Available: https://doi.org/10.2312/stag.20201241
[23] S. Loeng, "Self-Directed Learning: A Core Concept in Adult Education," *Education Research Int'l.*, no. 1, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1155/2020/3816132
[24] N. Ramli, P. Muljono, and F. M. Afendi, "External Factors, Internal Factors and Self-Directed Learning Readiness," *J. Edu. and e-Learning*, vol. 5, no. 1, p. 37–42, Jul. 2019. [Online]. Available: https://www.asianonlinejournals.com/index.php/JEELR/article/view/20
[25] T. H. Morris, "Four Dimensions of Self-Directed Learning: A Fundamental Meta-Competence in a Changing World," *Adult Education Quarterly*, vol. 74, no. 3, pp. 236–254, 2024. [Online]. Available: https://doi.org/10.1177/07417136231217453
[26] J. D. Robinson and A. M. Persky, "Developing Self-Directed Learners," *American J. of Pharmaceutical Edu.*, vol. 84, no. 3, p. 847512, 2020.
[27] S. L. Boyer, D. R. Edmondson, A. B. Artis, and D. Fleming, "Self-directed Learning: A Tool for Lifelong Learning," *Journal of Marketing Education*, vol. 36, no. 1, pp. 20–32, 2014.
[28] C. Lai, Q. Chen, Y. Wang, and X. Qi, "Individual Interest, Self-regulation, and Self-directed Language Learning with Technology Beyond the Classroom," *Br. J. Educ. Technol.*, vol. 55, no. 1, pp. 379–397, 2024. [Online]. Available: https://doi.org/10.1111/bjet.13366
[29] S. K. Aggarwal and M. S. Kumar, "Debuggers for Programming Languages," in *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2002, pp. 295–328. [Online]. Available: https://doi.org/10.1201/9781420040579.ch9
[30] M. A. Klimushenkova and P. M. Dovgalyuk, "Improving the Performance of Reverse Debugging," *Program. Comput. Softw.*, vol. 43, no. 1, pp. 60–66, 2017. [Online]. Available: https://doi.org/10.1134/S0361768817010042
[31] A. Savidis and V. Tsiatsianas, "Implementation of Live Reverse Debugging in LLDB," *CoRR*, vol. abs/2105.12819, 2021. [Online]. Available: https://arxiv.org/abs/2105.12819
[32] G. Pothier, É. Tanter, and J. M. Piquer, "Scalable Omniscient Debugging," in *Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. , 2007, pp. 535–552. [Online]. Available: https://doi.org/10.1145/1297027.1297067
[33] G. Pothier and É. Tanter, "Extending Omniscient Debugging to Support Aspect-Oriented Programming," in *Symposium on Applied Computing (SAC)*. , 2008, pp. 266–270. [Online]. Available: https://doi.org/10.1145/1363686.1363753
[34] M. van 't Riet, H. Yin, and C. Bockisch, "The Potential of Omniscient Debugging for Aspect-Oriented Programming Languages," in *Workshop on Comprehension of Complex Systems (CoCoS)*. , 2013, pp. 13–16. [Online]. Available: https://doi.org/10.1145/2451592.2451597
[35] C. Thiede, M. Taeumel, and R. Hirschfeld, "Time-Awareness in Object Exploration Tools: Toward In Situ Omniscient Debugging," in *SIGPLAN Onward!* , 2023, pp. 89–102. [Online]. Available: https://doi.org/10.1145/3622758.3622892
[36] K. Shimari, T. Ishio, T. Kanda, and K. Inoue, "Evaluating the Effectiveness of Size-Limited Execution Trace with Near-omniscient Debugging," *Sci. Comput. Program.*, vol. 236, p. 103117, 2024. [Online]. Available: https://doi.org/10.1016/j.scico.2024.103117
[37] P. Dovgalyuk, "Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging," in *16th European Conf. on Software Maintenance and Reengineering (CSMR)*. Computer Society, 2012, pp. 553–556. [Online]. Available: https://doi.org/10.1109/CSMR.2012.74
[38] N. Honarmand and J. Torrellas, "Replay debugging: Leveraging record and replay for program debugging," in *41st Int'l. Symposium on Computer Architecture (ISCA)*. Computer Society, 2014, pp. 445–456. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853229
[39] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering Record and Replay for Deployability," in *USENIX Annual Technical Conf.* USENIX Association, 2017, pp. 377–389. [Online]. Available: https://www.usenix.org/Conf./atc17/technical-sessions/presentation/ocallahan
[40] G. Malnati, C. M. Cuva, and C. Barberis, "JThreadSpy: Teaching Multithreading Programming by Analyzing Execution Traces," in *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*. , 2007, pp. 3–13. [Online]. Available: https://doi.org/10.1145/1273647.1273650
[41] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting Sequence Diagram from Execution Trace of Java Program," in *Int'l. Workshop on Principles of Softw. Evolution (IWPSE)*, 2005, pp. 148–154. [Online]. Available: https://doi.org/10.1109/IWPSE.2005.19
[42] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk, "Execution Trace Analysis Through Massive Sequence and Circular Bundle Views," *J. Syst. Softw.*,

vol. 81, no. 12, pp. 2252–2268, 2008. [Online]. Available: https://doi.org/10.1016/j.jss.2008.02.068

[43] B. Cornelissen, A. Zaidman, A. van Deursen, and B. V. Rompaey, "Trace Visualization for Program Comprehension: A Controlled Experiment," in *Int'l. Conf. on Program Comprehension (ICPC)*, 2009, pp. 100–109. [Online]. Available: https://doi.org/10.1109/ICPC.2009.5090033

[44] S. Jayaraman, B. Jayaraman, and D. Lessa, "Compact Visualization of Java Program Execution," *Softw. Pract. Exp.*, vol. 47, no. 2, pp. 163–191, 2017. [Online]. Available: https://doi.org/10.1002/spe.2411

[45] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring, "Live trace visualization for comprehending large software landscapes: The ExplorViz approach," in *First Working Conf. on Software Visualization (VISSOFT)*. Computer Society, 2013, pp. 1–4. [Online]. Available: https://doi.org/10.1109/VISSOFT.2013.6650536

[46] A. Savidis and N. Koutsopoulos, "Interactive Object Graphs for Debuggers with Improved Visualization, Inspection and Configuration Features," in *7th Int'l. Symposium on Advances in Visual Computing (ISVC)*, 2011, pp. 259–268. [Online]. Available: https://doi.org/10.1007/978-3-642-24028-7_24

[47] P. Gralka, C. Schulz, G. Reina, D. Weiskopf, and T. Ertl, "Visual Exploration of Memory Traces and Call Stacks," in *Working Conf. on Software Visualization (VISSOFT)*. , 2017, pp. 54–63. [Online]. Available: https://doi.org/10.1109/VISSOFT.2017.15

[48] C. Scully-Allison and K. E. Isaacs, "Evaluating Communication Pattern Representations in Execution Trace Gantt Charts," in *Working Conf. on Software Visualization (VISSOFT)*. , 2024, pp. 1–11. [Online]. Available: https://doi.org/10.1109/VISSOFT64034.2024.00011

[49] M. Weninger, L. Makor, and H. Mössenböck, "Memory Leak Visualization using Evolving Software Cities," *Softwaretechnik-Trends, Symposium on Software Performance (SSP)*, vol. 39, no. 4, pp. 44–46, 2019. [Online]. Available: https://fb-swt.gi.de/fileadmin/FB/SWT/Softwaretechnik-Trends/Verzeichnis/Band_39_Heft_4/SSP2019_Weninger.pdf

[50] ——, "Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor," in *Working Conf. on Software Visualization (VISSOFT)*, 2020, pp. 110–121. [Online]. Available: https://doi.org/10.1109/VISSOFT51673.2020.00017

[51] ——, "Heap Evolution Analysis Using Tree Visualizations," *Softwaretechnik-Trends, Symposium on Software Performance (SSP)*, vol. 40, no. 3, pp. 64–66, 2020. [Online]. Available: https://fb-swt.gi.de/fileadmin/FB/SWT/Softwaretechnik-Trends/Verzeichnis/Band_40_Heft_3/SSP2020_Weninger_Makor.pdf

[52] J. Sorva, V. Karavirta, and L. Malmi, "A Review of Generic Program Visualization Systems for Introductory Programming Education," *Trans. Comput. Educ.*, vol. 13, no. 4, pp. 15:1–15:64, 2013. [Online]. Available: https://doi.org/10.1145/2490822

[53] R. Oechsle and T. Schmitt, "JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)," in *Software Visualization*, ser. Lecture Notes in Computer Science, vol. 2269, 2001, pp. 176–190. [Online]. Available: https://doi.org/10.1007/3-540-45875-1_14

[54] M. Ben-Ari, N. Myller, E. Sutinen, and J. Tarhio, "Perspectives on Program Animation with Jeliot," in *Software Visualization, Int'l. Seminar Dagstuhl Castle*, ser. Lecture Notes in Computer Science, vol. 2269, 2001, pp. 31–45. [Online]. Available: https://doi.org/10.1007/3-540-45875-1_3

[55] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari, "Visualizing programs with Jeliot 3," in *Working Conf. on Advanced Visual Interfaces (AVI)*, 2004, p. 373–376. [Online]. Available: https://doi.org/10.1145/989863.989928

[56] S. M. Cisar, R. Pinter, and D. Radosav, "Effectiveness of Program Visualization in Learning Java: a Case Study with Jeliot 3," *Int. J. Comput. Commun. Control*, vol. 6, no. 4, pp. 668–680, 2011. [Online]. Available: https://doi.org/10.15837/ijccc.2011.4.2094

[57] M. Ben-Ari, R. Bednarik, R. B. Levy, G. Ebel, A. Moreno, N. Myller, and E. Sutinen, "A Decade of Research and Development on Program Animation: The Jeliot Experience," *J. Vis. Lang. Comput.*, vol. 22, no. 5, pp. 375–384, 2011. [Online]. Available: https://doi.org/10.1016/j.jvlc.2011.04.004

[58] J. Moons and C. De Backer, "The Design and Pilot Evaluation of an Interactive Learning Environment for Introductory Programming Influenced by Cognitive Load Theory and Constructivism," *Comput. Educ.*, vol. 60, no. 1, p. 368–384, jan 2013. [Online]. Available: https://doi.org/10.1016/j.compedu.2012.08.009

[59] J. Cross, D. Hendrix, and L. Barowski, "Combining Dynamic Program Viewing and Testing in Early Computing Courses," in *Int'l. Computer Software and Applications Conf. (COMPSAC)*, 08 2011, pp. 184 – 192.

[60] "Java Visualizer," https://cscircles.cemc.uwaterloo.ca/java_visualize, retrieved: 2025-05-23.

[61] "Python Tutor," https://pythontutor.com/, retrieved: 2025-05-23.

[62] M. Adamaszek, P. Chrzastowski-Wachtel, and A. Niewiarowska, "VIPER, a Student-Friendly Visual Interpreter of Pascal," in *Int'l. Conf. on Informatics in Secondary Schools - Evolution and Perspectives (ISSEP)*, ser. Lecture Notes in Computer Science, vol. 5090. Springer, 2008, pp. 192–203. [Online]. Available: https://doi.org/10.1007/978-3-540-69924-8_18

[63] F. J. Almeida-Martínez, J. Urquiza-Fuentes, and J. Á. Velázquez-Iturbide, "VAST: Visualization of Abstract Syntax Trees Within Language Processors Courses," in *Symposium on Software Visualization*. , 2008, pp. 209–210. [Online]. Available: https://doi.org/10.1145/1409720.1409759

[64] F. J. Almeida-Martínez and J. Urquiza-Fuentes, "Syntax trees visualization in language processing courses," in *Int'l. Conf. on Adv. Learning Technologies (ICALT)*. Computer Society, 2009, pp. 597–601. [Online]. Available: https://doi.org/10.1109/ICALT.2009.158

[65] F. J. Almeida-Martínez, J. Urquiza-Fuentes, and J. Á. Velázquez-Iturbide, "Visualization of Syntax Trees for Language Processing Courses," *J. Univers. Comput. Sci.*, vol. 15, no. 7, pp. 1546–1561, 2009. [Online]. Available: https://doi.org/10.3217/jucs-015-07-1546

[66] H. Dieterichs, "Debug Visualizer," https://marketplace.visualstudio.com/items?itemName=hediet.debugvisualizer, 2024.

[67] A. Jain, A. Goyal, and P. Chakraborty, "PPVT: A Tool to Visualize Predictive Parsing," *Inroads*, vol. 8, no. 1, pp. 43–47, 2017. [Online]. Available: https://doi.org/10.1145/3002136

[68] G. Costagliola, M. D. Rosa, V. Fuccella, and M. Minas, "ParVis: A Visual Tool for Exploring Parser Execution Traces," in *Int'l. Conf. on Advanced Visual Interfaces (AVI)*. , 2020, pp. 62:1–62:5. [Online]. Available: https://doi.org/10.1145/3399715.3399853

[69] ——, "Visual Exploration of Visual Parser Execution," *Multim. Tools Appl.*, vol. 81, no. 1, pp. 299–317, 2022. [Online]. Available: https://doi.org/10.1007/s11042-021-10624-6

[70] F. DeRemer, "Lexical Analysis," in *Compiler Construction, An Advanced Course, 2nd ed*, ser. Lecture Notes in Computer Science, vol. 21. Springer, 1976, pp. 109–120. [Online]. Available: https://doi.org/10.1007/3540069585_49

[71] Dr. Garbage Community, "Bytecode Visualizer," https://marketplace.eclipse.org/content/bytecode-visualizer, 2022.

[72] T. Dobravec, "Selected Tools for Java Class and Bytecode Inspection in the Educational Environment," *Open Comput. Sci.*, vol. 11, no. 1, pp. 43–50, 2021. [Online]. Available: https://doi.org/10.1515/comp-2020-0170

[73] M. J. Alam, M. T. Goodrich, and T. Johnson, "J-Viz: Finding Algorithmic Complexity Attacks via Graph Visualization of Java Bytecode," in *Symposium on Visualization for Cyber Security (VizSec)*. Computer Society, 2016, pp. 1–8. [Online]. Available: https://doi.org/10.1109/VIZSEC.2016.7739575

[74] C. Redestad, "Bytestacks," https://cr.openjdk.org/∼redestad/bytestacks/bytecode-startup-analysis.html.

[75] B. Gregg, "The Flame Graph," *ACM Queue*, vol. 14, no. 2, p. 10, 2016. [Online]. Available: https://doi.org/10.1145/2927299.2927301

[76] ——, "The Flame Graph," *Commun. ACM*, vol. 59, no. 6, pp. 48–57, 2016. [Online]. Available: https://doi.org/10.1145/2909476

[77] S. Schäckeler, W. Shang, and R. Davis, "Compiler Optimization Pass Visualization: The Procedural Abstraction Case," *Trans. Comput. Educ.*, vol. 9, no. 2, pp. 14:1–14:13, 2009. [Online]. Available: https://doi.org/10.1145/1538234.1538241

[78] R. Koitz-Hristov, F. Mandl, and F. Wotawa, "VisOpt - Visualization of Compiler Optimizations for Computer Science Education," in *Technical Symposium on Computer Science Education (SIGCSE)*. , 2025, pp. 603–609. [Online]. Available: https://doi.org/10.1145/3641554.3701832

[79] H. Lim and S. G. Kobourov, "Visualizing JIT Compiler Graphs," in *29th Int'l. Symposium on Graph Drawing and Network Visualization (GD)*, ser. Lecture Notes in Computer Science, vol. 12868. Springer, 2021, pp. 138–146. [Online]. Available: https://doi.org/10.1007/978-3-030-92931-2_10

[80] T. Herber and M. Weninger, "Trace-Based Bytecode Interpreter Visualization for Compiler Construction Education - Artifact (Source Code, Video, Instructions, Demo Traces (Version 1.1))," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.16899998