`JastAdd Bridge`: Interfacing reference attribute grammars with editor tooling

Johannes Hardt
Lund University
jo8482ha-s@student.lu.se

Dag Hemberg
Lund University
da6673he-s@student.lu.se

Abstract

Nowadays, programmers expect rich editor support when writing code, such as type hints, code completions, and warnings. While this is present in mainstream languages and editors, lacking such tooling can be a pain point when developing your own language. Our paper proposes an extension for the `VS Code` editor, `JastAdd Bridge`, that acts as a translation layer between languages developed with the `JastAdd` framework and the language server protocol (LSP). We find that it facilitates rapid integration with editor tooling while preserving flexibility in the way language authors choose to interact with the programming environment.

Keywords: language, language server, lsp, jastadd, codeprober, vscode

1 Introduction

IDEs (*integrated development environments*) are tools used by many programmers. The purpose of an IDE is to help developers efficiently write and debug code written in most languages, providing debugging tools, diagnostics, quick-fixes and refactoring options for common antipatterns, among others. These functionalities are often not provided by the editor itself, but by a *language server*, which compiles and analyzes code and gives relevant information back to the editor.

However, not all languages have an implementation of a dedicated language server. Newer, smaller languages often don't have enough active users or developers who have enough spare time to integrate a fully-featured server. Those developing their own languages and compilers from scratch face the same issue, since the language server is often a separate challenge to the language itself.

1.1 Motivation

The $M \times N$ problem According to the StackOverflow developer survey [1], almost 74% of developers surveyed regularly used the IDE named *Visual Studio Code* (VS Code). VS Code features extension support, giving developers an easy way to add support for new languages. However, the same survey also shows widespread support for other editors such as `IntelliJ IDEA` and `Vim`, both of which support plugins or extensions of their own. From this, an issue arises: given M different IDEs with plugin support and N different languages, $M \times N$ separate integrations are needed, a number which quickly grows as more editors and languages are developed.

Thankfully, there is a solution to this problem: the *language server protocol* [2], or LSP. LSP provides a standardized way for language servers and editors to easily communicate with each other, abstracting away most or all implementation details specific to any one editor language server. Instead of integrating with one specific editor, language servers can implement their plugins against LSP. Likewise, IDE developers can integrate against LSP without worrying about language-specific details. This brings the total number of integrations required given M IDEs and N language servers down to $M + N$.

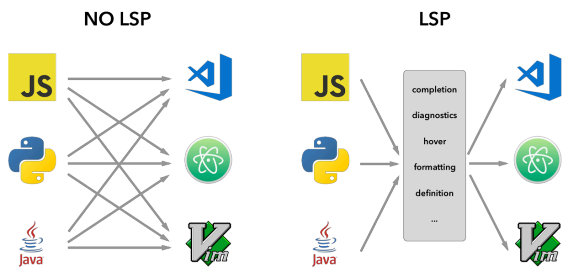


Figure 1. $M \times N$ integrations turning into $M + N$ through the use of LSP. Source: [13]

Language Server Protocol Developed by Microsoft, LSP uses JSON-RPC-based messaging to send instructions and information between the editor and the server. The client (editor) continuously sends information to the server about what the user is doing, e.g typing, saving, or requesting some action be done such as quickfixing, and the server responds with some appropriate action (updating autocomplete suggestions, performing the quickfix, updating error diagnostics, etc) or message. A diagram demonstrating an example of this is shown in Figure 2.

Course paper, EDAN70, Lund University, Sweden
January 25, 2024.

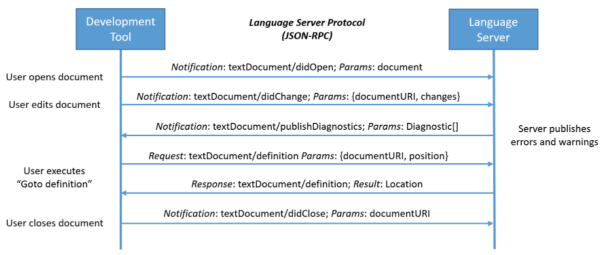

Development Tool
Language Server Protocol (JSON-RPC)
Language Server
User opens document
Notification: textDocument/didOpen; Params: document
User edits document
Notification: textDocument/didChange; Params: {documentURI, changes}
Notification: textDocument/publishDiagnostics; Params: Diagnostic[]
Server publishes errors and warnings
Request: textDocument/definition Params: {documentURI, position}
User executes "Goto definition"
Response: textDocument/definition; Result: Location
User closes document
Notification: textDocument/didClose; Params: documentURI

Figure 2. Diagram of requests and responses sent back and forth between the client and server through LSP. Source: [3]

LSP by itself is, as the name implies, a protocol, and as such requires some actual implementation in order to be used. One such implementation is `LSP4j`, which provides an API to use LSP features in Java.

JastAdd While LSP makes it easier to integrate language servers and editors, the landscape of programming is constantly evolving, and new languages are constantly being developed. Developing a new language is often analogous to writing a compiler for that language using a meta-compiler, or compiler-compiler. Providing LSP support on a more general level (e.g., from within the meta-compilation system) further eases the process of integrating a new language with an editor and provides a pleasant experience for developers in that language.

For Java, one such meta-compilation system is `JastAdd`. `JastAdd` features support for Reference Attribute Grammars (RAGs), used for conveniently extending compilers with functionality or language constructs without compromising ease of computation. Using `JastAdd`, users can synthesize methods onto existing classes or nodes in an AST, and refer to those nodes in other synthesized methods. This allows for features such as static analysis or code generation to be added without manually modifying the source code of a program. Users can also divide their code into modules called *aspects*, where similar or related functionality is often grouped together.

1.2 Problem description

As of now, there isn't any way to quickly integrate languages or tools specified with `JastAdd` with code editors. An existing tool, *`CodeProber`* [14], allows for inspection of the generated AST, provided you give it the path to a `JastAdd` compiler. While this is helpful for developing a language with `JastAdd`, it doesn't provide the tooling that is expected in modern software development. Another drawback is that users cannot choose their editor, since `CodeProber` provides its own web-based editor.

2 Solution

We propose a VS Code extension that provides *language server protocol* (LSP) [2] support in `JastAdd`-based languages. This extension can be configured with a path to a `.jar` file built with `JastAdd`, and can then provide editor interactions for the provided tool. The language server acts as an API between common LSP interactions and a set of `JastAdd` attributes. These attributes allow the writer of a `JastAdd` tool to specify how different interactions should work with their specific language.

One goal of our extension is to significantly decrease the amount of effort for enabling editor interactions for your language. For smaller projects, the amount of effort writing the LSP server and editor extensions might not be worth the benefits. Similar approaches have been tried before, for example, the MagpieBridge framework [12]. By providing an abstraction layer (similar to ours) on top of the LSP, it allows simplified integration of static analysis tools with editors.

2.1 Implementation

Supporting the LSP for an editor consists of implementing two separate parts: a *client*, i.e. an application that interfaces with the code editor (`VS Code` for our extension), and a *server*, as shown in Figure 3. The client is closely coupled with the host editor and can access functionality specific to the editor. In our case, we show prompts when the extension is misconfigured, store settings in the `VS Code` settings page, and launch the server when VS Code detects the appropriate extension.

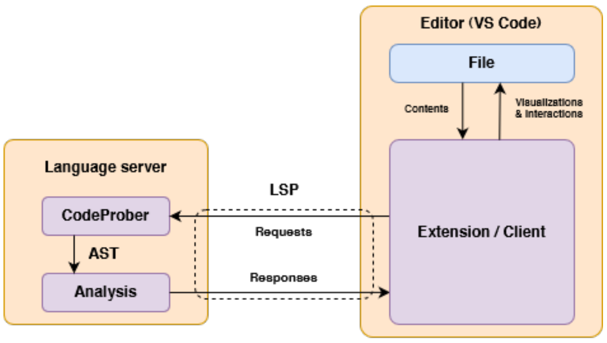


Figure 3. Architectural overview of the client-server structure of the extension

The server is more general and should work with other clients. It receives requests from the client over some kind of connection (e.g. sockets, pipes, `stdin` / `stdout`), handles them, and sends back a response. Our extension uses the *`LSP4j`* implementation of the LSP [9]. The library provides classes with methods that can be overridden to handle incoming requests.

Code analysis is delegated to the `CodeProber` library mentioned above. Using reflection, it can build an AST by invoking the user-provided compiler on some source code.

2.2 Attributes

The extension uses some predetermined `JastAdd` attributes to allow users a way of configuring editor interactions for their use cases. To avoid name clashes, all attributes are prefixed with `lsp_`.

The supported interactions of this extension are as follows:

- Hover
- Diagnostics
- Quickfixes ("*Code actions*")
- Go to definition
- Run lens / "`Run ▶`" button ("*Code lens*")

When implementing some interactions, the extension needs more information than can be provided by a primitive type (e.g. both the message and severity for diagnostics). Therefore, some attributes are expected to accept or return objects that fit a certain shape. We describe the attributes and their APIs below. All images shown in the following subsections are results of LSP integrations created using the extension.

2.2.1 Hover

Hover support (the tooltips shown when hovering the mouse pointer over some token, as shown in Figure 4) is provided using the `String ASTNode.lsp_hover()` attribute. The `ASTNode` signature can also be replaced with a more specific node type, such as `IdUse` or `FuncDecl`.

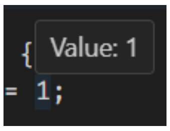


Figure 4. Tooltip shown after hovering over a symbol

When the user hovers over some character in a file, a `textDocument/hover` request, containing information such as the hover position and file name, is sent from the client to the server. Using the position provided in the request, the server finds the AST node at that position and attempts to invoke the `lsp_hover()` method on it. The resulting string, if the attempt succeeds, is then sent back to the client for interpretation and displaying. If the attempt does not succeed, i.e., the attribute isn't implemented on the node at the given position, then no tooltip is shown in the client.

2.2.2 Diagnostics

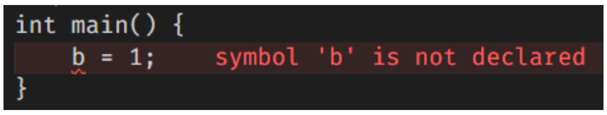
Diagnostics, colloquially known as "squiggly lines", are a highly useful way to quickly provide information at a glance about errors, warnings, hints or info regarding the code.

To provide support for diagnostics relating to static code analysis for languages built using `JastAdd`, the user must implement the attribute `Set<Diagnostic> Program.lsp_diagnostics()`[1]. `Diagnostic` is an interface that should contain information about the severity, location, and a message to the user. To support a wide range of implementations, we only require that some getters are implemented to access this information, which are shown in Listing 1.

```
interface Diagnostic {
  String message();
  int severity();
  int startLine();
  int startColumn();
  int endLine();
  int endColumn();
}
```

Listing 1. Methods that the extension assumes are implemented when using the `lsp_diagnostics()` attribute.

`message()` is the actual message provided to the user about what issue the diagnostic represents. The four methods `startLine()`, `startColumn()`, `endLine()` and `endColumn()` combine to specify where in the file the diagnostic has occurred, and `severity()` signifies what type of diagnostic this is (error, warning, info or hint). An example of a diagnostic can be seen in Figure 5.



```
int main() {
    b = 1;     symbol 'b' is not declared
}
```

Figure 5. Diagnostic with severity indicating an error

When a user saves, opens, or closes a document, the client notifies the server of this. The server responds by telling the client to refresh or clear its own list of diagnostics. The client sends a `textDocument/diagnostic` request to the server, making the server invoke the `lsp_diagnostics()` attribute, from which it can build a `DocumentDiagnosticReport` to send back to the client.

Syntax errors Diagnostics coming from static analysis of code, like the one shown in Figure 5, rely on the AST having been built correctly. However, in many cases where diagnostics are useful, such as indicating missing semicolons

[1]the names `Diagnostic` and `Program` are not definite, but instead refer to an interface and the root node of the AST, respectively. Users can name these anything, provided that all abstract methods are named as shown in the interface and that the method is implemented on the root node. This also applies to the `Edit` interface shown later in the paper, as well as all future references to a `Program` node.

or unbalanced parentheses, this is not the case, since these errors all stop the compiler from building the AST.

To solve this issue, the extension also uses `CodeProber` to parse messages sent to `stdout` or `stderr` from the compiler during the scanning or parsing stages and convert them into `Diagnostics`, which are then added to the list of `Diagnostics` making up the `DocumentDiagnosticReport`.

2.2.3 Quickfixes

Quickfixes (`textDocument/codeAction` in LSP) are actions the user can take to quickly resolve diagnostics, either through text insertion, replacement, or refactoring. In VS Code, these usually appear as an icon of a lightbulb when the cursor is placed within a range in which a quickfix would affect the code. Pressing this lightbulb or using the dedicated keyboard shortcut (`Ctrl / Cmd + .` by default in `VS Code`) brings up a menu where all quickfixes or actions are displayed, and choosing one of these actions executes it without the need for further user input.

Providing quickfix support in `JastAdd`-based languages does not mean implementing a new attribute on AST nodes, but rather extending[2] the definition of the `Diagnostic` interface, as well as implementing a new `Edit` interface, as shown in Listing 2. This also means that diagnostics need to be fully implemented in order for quickfixes to function properly.

```
interface Diagnostic {
  // previously declared methods omitted
  Set<Edit> edits();
  String codeActionTitle();
}

interface Edit {
  String replacement();
  int startLine();
  int startColumn();
  int endLine();
  int endColumn();
}
```

Listing 2. Additional methods in the `Diagnostic` and `Edit` interfaces assumed by the extension to be implemented for quickfixes.

[2] In our extension, quickfixes are an optional component of diagnostics represented as `Set<Edit>`. This differs slightly from the actual implementation of `codeActions` in LSP, where each action instead contains a list of related diagnostics which would be resolved if the action were to be applied. Since one of the focal points of our extension is ease of use, the decision was made to remove some of the flexibility provided by the LSP standard in favor of simplicity.

The `Edit` interface provides information about what should happen when the quickfix is applied. Similarly to `Diagnostic`, the `Edit` interface should contain the methods `startLine()`, `startColumn()`, `endLine()` and `endColumn()` to signify the text to be replaced. `replacement()` is the text to be inserted in place of the text affected by the range.

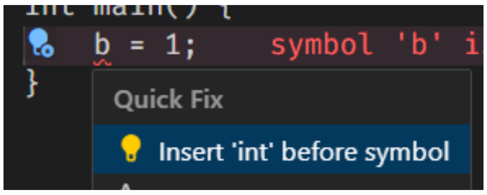
The `codeActionTitle()` attribute in `Diagnostic` is the message displayed when showing available quickfixes for some diagnostic. The `edits()` attribute is the set of `Edits` which are applied when resolving the diagnostic. In many cases, a single `Edit` can be enough to resolve an issue—but in more complex cases it can still be useful for a quickfix to contain multiple `Edits`. For example, in a language with a module structure, one quickfix could be to replace qualified usage with an import, see Listing 3.

Before	**After**
`// some code`	`import Foo.Bar.baz;`
`Foo.Bar.baz();`	`// some code`
	`baz();`

Listing 3. Possible application of using multiple `Edits` in a quickfix

This quickfix would require two `Edits`: one for removing the path from the function call, and one for inserting the `import` statement at the top of the file.

Unline diagnostics, quickfix (`textDocument/codeAction`) requests are sent from the client to the server whenever the cursor changes position. The server then manually filters out all available quickfixes from the cached AST to the ones applicable at that cursor position, after which it sends them back to the client to interpret.



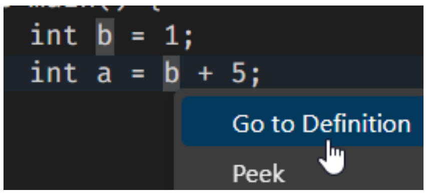
```
int main() {
    b = 1;     symbol 'b' i
}
```

Figure 6. Example of a quickfix

2.2.4 Go to definition

"Go to definition" actions make it possible for users to quickly jump to declarations or definitions of variables, functions, or other language constructs from some usage elsewhere in the program. With our extension, users can achieve this functionality by implementing the `ASTNode ASTNode.lsp_definition()` method.

A typical implementation of this could be some `VariableUse` node, representing the usage of a previously declared variable, implementing the attribute, and returning the AST node representing the declaration of that variable. An example demonstrating what this looks like in VS Code is shown in Figure 7.



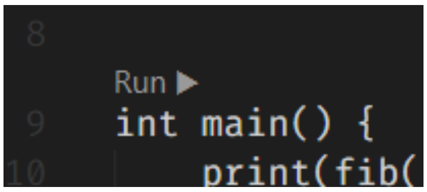
```
int b = 1;
int a = b + 5;
```

Figure 7. "Go to definition" implementation

`textDocument/definition` requests are sent either once the user holds `Ctrl` or `Cmd` (when using Windows / Linux or Mac, respectively) and hovers over some symbol, or invokes the action manually by pressing the dedicated keyboard shortcut (`F12` by default on VS Code) or right-clicking the text and pressing "Go to definition". The server then finds the node at that position and attempts to invoke the `lsp_definition` method on that node. If successful, it sends a `Location`, used to represent a position in a text file, based on the position of the node returned from the method, to the server.

2.2.5 Run lens

The Run lens is a specific application of the `textDocument/codeLens` LSP request, meant to evaluate the written program from some given entry point when activated. To use this functionality, users must implement two methods in `JastAdd: ASTNode Program.lsp_main()`, for locating the actual entry point, and `public void Program.lsp_run()` for determining what happens when the run lens is activated.



```
int main() {
    print(fib(
```

Figure 8. Run lens

Analogously to the two methods, there are two different LSP requests handling this functionality. To show the run lens, the `textDocument/codeLens` request is sent on each document save, making the server attempt invocation of `lsp_main()` on the AST root node. If successful, the position of the entry point node is computed and sent back to the client. Pressing the "Run ▶" button invokes the sending of a `workspace/executeCommand` request for a command called `jastaddBridge.run`, which the server receives, upon which it attempts to invoke `lsp_run()` on the root node.

2.3 Extension configurations

Our extension also features some additional features and configurations:

- "Partial" and "purge" options for caching to help in debugging
- "Hot reloading", or automatically updating the used compiler once rebuilt
- 3 different tracing options for debugging ("verbose", "messages" and "off")
- The ability to override the `JAVA_HOME` environment variable only within the extension for supporting different Java versions

We've tried to ensure that the extension works as expected in different configurations. Errors are clearly reported both as native `VS Code` notifications with action buttons and in the debug log with additional information. This is both for unexpected runtime errors and common misconfigurations, such as having an incorrect `JAVA_HOME` environment variable set.

3 Evaluation

3.1 Method

We chose to evaluate our solution by writing LSP integrations for existing tools using the features the extension provides. These tools included 3 `SimpliC` compilers, built in the EDAN65 Compilers course at LTH, as well as `CalcRAG`, a small language used in the EDAN65 course as an introduction to reference attribute grammars and static analysis.

To aid in the integration process, we also used a library with pre-built `Diagnostic` and `Edit` classes with useful helper methods, which is also available to other users of our extension.

3.2 Result

For each compiler we evaluated, we implemented a `Codeprober_parse` method in its main class, needing around 10 lines of code. The reason for this was to provide an separate entry point for `CodeProber` to avoid clashing with other output from the "regular" main method when parsing syntax errors[3] (see [4]). Also required was a sim-

[3]Upon later reflection, this addition was likely not needed, since `CodeProber` falls back to using the `main` method provided that there is an `Object CodeProber_root_node` or `Object DrAST_root_node` attribute in the main class.

ple class for handling parsing errors, containing a message string and start and end positions. This class was the same in all the tools we used in the evaluation, and consisted of about 15 lines of code.

Once these features were implemented, integrating LSP features into the compilers required about one to two lines of code per feature for "hover", "go to definition" and "run lens", while diagnostics required about 4 to 5 lines of code per reported diagnostic. Implementing diagnostics simply meant copying the already-implemented error analysis contributions and adapting them to fit the provided interface for `JastAdd` LSP diagnostics. Using our pre-built library for `Diagnostic` and `Edit` classes also meant that implementing quickfixes only required an additional few lines per implemented diagnostic report.

3.3 Discussion

The evaluation shows a noteworthy reduction in required code compared to the alternative approach of individually supporting each programming language, as that alternative would entail creating a slightly less generalized version of the entire project, tailored to each specific language. We feel our solution is more accessible and easier to comprehend than the intricacies that come with LSP, especially given the lack of comprehensive documentation for both LSP and `LSP4j`.

However, as the authors of the extension we inherently have a deeper understanding of its workings from the outset compared to the average user. While efforts have been made to make the extension easily accessible through clear documentation in the extension description, there remains a concern about the potential lack of clarity for users approaching the extension for the first time, especially since no formal user study was conducted to assess the practicality and usability of the extension.

An additional point of concern is the absence of feedback when misspelling or using the wrong name for an attribute. With users only realizing something went wrong when the intended integration is not functioning as expected without any clear indication as to the cause, successful integration could be perceived as more challenging.

In an attempt to address usability concerns, external individuals were consulted during the development of the extension. Their preference for our extension over the similarly functioning `MagpieBridge` was noted; however, it is crucial to acknowledge that this observation was made during a guided implementation session, and thus was not included in the formal evaluation process.

4 Related work

While LSP is a solution to the $M \times N$ problem described above, it may be too complex to implement for smaller projects. In [10], the authors shed some light on the engineering practices used in LSP implementations, by studying the source code of 30 open-source LSP codebases. While they provide valuable concerns and detailed practices for efficiently implementing LSP, they also note that "further modularizing LSPs is still an open research problem".

Attempts have been made to build upon the protocol to provide a more lightweight interface. In [15], the authors build upon the Truffle framework [5] to add API methods that facilitate integration with LSP. Similar to `JastAdd`, implementing a language with Truffle entails generating an abstract syntax tree, where meta-information can be added in the form of "instrumentation nodes" [16]. Truffle and the LSP extension proposed by [15] are aimed at providing language support and runtime analysis for dynamic languages, which come with some drawbacks. The authors note that the static information available in dynamic, untyped languages is very limited. To obtain more information about some source file, the approach they take is to execute parts of it. A complication that arises is that the analysis must now be sandboxed since users do not expect their editor to hang if it reaches some code that doesn't terminate, for example [15].

Our approach instead builds upon the `JastAdd` meta-compilation system, which uses reference attribute grammars, instead of instrumentation nodes to annotate nodes in the AST with additional information [11]. This approach is flexible, in that compiler writers can both modify behavior and add language constructs, directly on the abstract syntax tree [6]. Using `JastAdd` allows our API to be small, as users only have to add some predefined JastAdd attributes to their compiler, with no need for dependencies. Another benefit is that restricting editor integrations to specific language constructs is simple since you choose where in the AST you add attributes such as `lsp_hover()`.

MagpieBridge is perhaps the project that is the most similar to ours [12]. It allows writers of static analysis tools to provide editor integrations using LSP, without implementing the protocol themselves. Like our implementation, MagpieBridge is written in Java and uses the `LSP4j` library to interface with LSP. The main difference is that while MagpieBridge is an abstraction upon LSP, users are still expected to bundle an extension with their analysis code and write some amount of boilerplate to set up a connection to their analysis server.

As mentioned, we don't contribute any source analysis - it is handled by the `CodeProber` library. `CodeProber` is a tool for visualizing *property probes* [14], initially built to support tools written with the `JastAdd` system. In our language server, it is used in three ways:

- Locating the AST node on a particular location in a source
- Extracting location information from an AST node
- Intercepting syntax errors in `stdout` or `stderr` to report as diagnostics

The graphical interface `CodeProber` provides can be helpful while developing a compiler with `JastAdd`, as well as while implementing the attributes to integrate with our language server. Having written some code in the editor, probes can be created by right-clicking on it, which reveals all overlapping AST nodes and the attributes defined on them.

5 Conclusion

We have developed a language server, `JastAdd Bridge`, that can be used for implementing editor support for languages created with the `JastAdd` meta-compilation system. The language server can communicate with any editor implementing the language server protocol, making it useful for a variety of tools. Additionally, we contribute an extension for the popular editor *Visual Studio Code*, which both integrates the editor with our language server and exposes some common configuration options of the server to the `VS Code` settings panel. Currently supported integrations allow users to:

- propagate both parsing and compilation errors to the editor interface "diagnostics"
- optionally add fixes to these errors that can be resolved by clicking the button "quick fixes"
- customize what information is shown when hovering over code "hover tooltips"
- execute a main function directly in the editor "run button"
- following the use of some variable to its declaration "go to definition"

Compiler developers using our extension specify which of these integrations they want to enable using special `JastAdd` attributes we have specified. These attributes are then read using runtime reflection when the server performs analysis. Extending some compilers (built with `JastAdd`) to support our editor integrations, we found that this approach is both flexible and requires a minimal amount of additional code.

5.1 Future work

Expansion to other editors One clear direction going forward for the project is support for additional editors with LSP support, such as `NeoVim`, `IntelliJ IDEA`, or `Atom`. The server aspect of the extension is intended to be fully modular, with the only requirement for support for other editors being new client-side code specific to each editor.

Some modifications to the language server would help make it more compatible with other editors. Currently, the server assumes that an editor implements all functionality that the server uses. This may not be the case, and LSP has a way for editors and language servers to negotiate what functionality will be used between them, based on what they both support (*capabilities*, see [7]). Furthermore, language servers commonly implement multiple "communication channels", to be compatible with more editors [8]. `JastAdd Bridge` currently only supports communication over a WebSocket.

Multi-file support At the time of writing, the extension only supports working in a single file at a time, meaning actions such as "go to definition" cannot be meaningfully used across files. As software projects often span multiple files, enhancing the extension to support more than a single file would extend its usefulness to larger and more complex compiler projects.

Responsiveness Currently, the language server reparses code when the opened file is saved. This approach was simple to implement but can feel somewhat unresponsive, as feedback such as syntax errors is not shown until the current file is saved. A better solution would be to incrementally update an internal cached version of the current document on every keystroke.

It is common for extensions that implement a "run button" / "run lens" to launch a new integrated terminal with the output of executing the current file. Our extension instead shows it in the `VS Code` "output" tab, which has some drawbacks.

1. The output tab can become cluttered, as parsing errors are also printed here.
2. The output tab isn't shown when pressing the run button, it instead has to be opened manually, which can give the impression that the code was not executed.

More LSP features The language server protocol supports a multitude of different editor features, and in our extension, only 5 of these have been implemented. For example, some features common among other language servers are semantic syntax highlighting, code completion, and renaming, all of which provide a better coding experience for the user.

6 Acknowledgments

- Anton Risberg Alaküla, for being an enthusiastic and incredibly helpful project supervisor, and for developing `CodeProber`, without which this project would not have been possible.
- Oskar Stenberg, Esbjörn Stenberg, Daniel Adu-Gyan, Max Sjölin, Elsa Cervetti Ogestad, and Patrik Gyllvin for providing their `SimpliC` compilers to us to help evaluate the usability of our program.
- Mark Lundager and Andreas Bergquist, for taking the time to try out implementing our extension into their EDAN70 project.

References

[1] Stack Overflow Developer Survey 2023. Retrieved November 17, 2023 from https://survey.stackoverflow.co/2023/

[2] Official Page for Language Server Protocol. Retrieved November 10, 2023 from https://microsoft.github.io/language-server-protocol/

[3] What is the Language Server Protocol?. Retrieved January 10, 2024 from https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/

[4] Lu-Cs-Sde/Codeprober. Retrieved January 8, 2024 from https://github.com/lu-cs-sde/codeprober/tree/6e9bc54f82609241fdd498b6565fffa43d837726

[5] Truffle Language Implementation Framework. Retrieved December 1, 2023 from https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/

[6] JastAdd.Org. Retrieved January 8, 2024 from https://jastadd.cs.lth.se/web/documentation/concept-overview.php

[7] LSP Specification - Capabilities. Retrieved January 10, 2024 from https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#capabilities

[8] LSP Specifications - Considerations. Retrieved January 10, 2024 from https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#implementationConsiderations

[9] 2023. Eclipse-Lsp4j/Lsp4j. Retrieved November 17, 2023 from https://github.com/eclipse-lsp4j/lsp4j

[10] Djonathan Barros, Sven Peldszus, Wesley K. G. Assunção, and Thorsten Berger. 2022. Editing Support for Software Languages: Implementation Practices in Language Server Protocols. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, October 23, 2022. ACM, Montreal Quebec Canada, 232–243. Retrieved December 1, 2023 from https://dl.acm.org/doi/10.1145/3550355.3552452

[11] Görel Hedin and Eva Magnusson. 2003. JastAdd—an Aspect-Oriented Compiler Construction System. *Science of Computer Programming* 47, 1 (2003), 37–58. Retrieved from https://www.sciencedirect.com/science/article/pii/S0167642302001090

[12] Linghui Luo, Julian Dolby, and Eric Bodden. 2019. MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs)*, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1–25. Retrieved from http://drops.dagstuhl.de/opus/volltexte/2019/10813

[13] Malintha Ranasinghe. Understanding the Language Server Protocol. Retrieved January 10, 2024 from https://medium.com/codex/understanding-the-language-server-protocol-lsp-5957a571bf0f

[14] Anton Risberg Alaküla, Görel Hedin, Niklas Fors, and Adrian Pop. 2022. Property Probes: Source Code Based Exploration of Program Analysis Results. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, November 29, 2022. ACM, Auckland New Zealand, 148–160. Retrieved November 10, 2023 from https://dl.acm.org/doi/10.1145/3567512.3567525

[15] Daniel Stolpe, Tim Felgentreff, Christian Humer, Fabio Niephaus, and Robert Hirschfeld. 2019. Language-Independent Development Environment Support for Dynamic Runtimes. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*, October 20, 2019. ACM, Athens Greece, 80–90. Retrieved December 1, 2023 from https://dl.acm.org/doi/10.1145/3359619.3359746

[16] Michael L. Van De Vanter. 2015. Building Debuggers and Other Tools: We Can "Have It All". In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, July 04, 2015. ACM, Prague Czech Republic, 1–3. Retrieved December 1, 2023 from https://dl.acm.org/doi/10.1145/2843915.2843917