Extending JastAdd Bridge: Bringing more features from LSP into JastAdd

Julia Karlsson
D20, Lund University, Sweden
ju3007ka-s@student.lu.se

Philip Sadrian
D20, Lund University, Sweden
ph8605sa-s@student.lu.se

***Abstract*—Many features exist in modern IDEs to improve the programming workflow of developers, such as code completion, go to definition, refactoring and symbol outlines of a text file. These features are often provided by language servers. However, implementing a language server is tedious, even though the Language Server Protocol exists to standardize the process. In this paper, we present continued work on the JastAdd Bridge (JAB) extension for Visual Studio Code (VS Code), thus expanding the translation layer between JastAdd defined languages and the Language Server Protocol.**

Our work resulted in the addition of symbol outline and code completion functionality. We found that JAB now seems to support most integral LSP functionality, while still being simple to implement through both a case study of integration with ExtendJ and a small user study.

***Keywords*—language server protocol, jastadd, jastadd bridge, ExtendJ**

I. Introduction

To simplify the implementation of IDE features, a protocol called *Language Server Protocol* (LSP) can be used. Most modern IDEs support the client side of the protocol. They connect to Language Servers in order to compute what to display in the IDE, such as the value of a variable when hovering over it. The abstraction layer provided by LSP allows a single server to be used in multiple IDEs, thus removing the need to develop a new server for each IDE. This is means LSP reduces the previous $m \times n$ problem to a $m + n$ problem instead, where m is the number of editors and n the number of languages [1].

The meta-compiler *JastAdd* [2] can be used to create compilers and static analysers, and is described as "[...] a safer and more powerful alternative to the Visitor pattern". JastAdd works by using *Reference Attribute Grammars* (RAG) [3], which are an extension to *Attribute Grammars* (AG) [4]. AGs is a method to declaratively define valued properties of symbols in a context-free grammar. A limitation of AGs is that it is impossible to reference non-local nodes in the AST, which means that other nodes cannot be used in calculations. This is instead addressed in RAGs [3].

Supporting LSP requires some implementation work, however, there are tools to facilitate the implementation of the protocol. One way to do this for languages generated with JastAdd is by using the tool called *JastAdd Bridge* (JAB) [5]. This enables a simpler way to support LSP functionality without having to write a brand new language server from scratch, saving development time. JAB translates JastAdd attributes to LSP interactions, by having the compiler author implement special, pre-defined attributes. This translation is done by implementing a server interface defined in LSP4J [6], which is a tool that can be used to develop a language server in Java.

In this paper we present work on extending the functionality of JAB to increase its usefulness for programming language developers wanting to support Language Server functionality.

To exemplify use of JAB with a feature-complete compiler and compare the results, we used *ExtendJ*. It is a Java compiler developed with JastAdd, designed to be extensible [7]. We also performed a small user study to further evaluate our additions.

Course paper, EDAN70, Lund University, Sweden
January 13, 2025

II. Background

The first iteration of JAB [5] implemented a subset of the functionality defined by the LSP. These were *hover tool-tips* over symbols, *error diagnostics*, *quick-fixes*, *go to symbol definitions*, and a *run lens* that enables running a program. That work was done as a part of the same course as this paper (*EDAN70 Project in Computer Science*), and had quite a limited development time. It offers a good foundation to continue development on as the underlying communication and structure of the extension itself can be reused.

As mentioned in Section I, the integration between a compiler and JAB is achieved by the compiler author implementing special JastAdd attributes. These are prefixed with `lsp_` to avoid name clashes. The initial version of JAB expected the following attributes to provide the supported functionality [5]:

```
syn String ASTNode.lsp_hover();
coll Set<Diagnostic> Program.lsp_diagnostics();
syn ASTNode ASTNode.lsp_definition()
syn ASTNode Program.lsp_main();
public void Program.lsp_run();
```

Fig. 1: All attributes that were implemented in the initial version of JastAdd Bridge

As a part of the first iteration of JAB [5] the `interop` library was created, defining the Java classes required by the attributes shown in Fig. 1. The library simplifies integration by allowing a compiler developer to import the package and use the needed classes, instead of writing classes from scratch that conform to the requirements of JAB. Some notable classes contained in `interop` are `Diagnostic` and `Edit`, that enable the LSP features of diagnostics and quick fixes.

III. Solution

To improve JastAdd Bridge, the features *Document Symbol Outline* and *Code Completion* were deemed useful for the end user. We considered other features, but chose to focus on expanding support of LSP functionality. For a listing of the new attributes introduced in this paper, see Fig. 2.

```
coll Set<Symbol> ASTNode.lsp_symbols();
syn Set<CompletionItem> ASTNode.lsp_completions();
```

Fig. 2: The special attributes that were added as part of the solution.

A. Document Outline

In order for a developer to get an overview of the contents of a file when writing a program, we added the possibility to generate a hierarchical document outline of the symbols in a document. Some examples of symbols are classes, functions and variables. This is used similarly to diagnostics and is easily implemented with a collection attribute in JastAdd [8]. The special attribute that the compiler needs to define is named `lsp_symbols()`, which returns a Java set `Set<Symbol>`. Consequently, to let a certain type of node appear in the outline, let it contribute a `Symbol` to `lsp_symbols()`. The class `Symbol` is defined in the `interop` library as:

```
class Symbol {
   String getName();
   Object getNode();
   int getKind();
   int getStartLine();
   int getStartColumn();
   int getEndLine();
   int getEndColumn();
}
```

The `getKind()` method is required to return a number between 1 and 26, and must match the LSP definition [9] of `SymbolKind` to render the expected document symbol icons in the outline. There is a static class in the `interop` library that contains all available types of symbols. They are mapped to the corresponding values of `SymbolKind` defined by the LSP specification. The method `getNode()` must return the AST Node that belongs to the symbol.

Using a tree-traversal algorithm in the server, JastAdd Bridge automatically detects whether there is a hierarchy in the document symbols or not, as can be seen in Fig. 3. The algorithm removes the need of manually specifying the hierarchy in JastAdd. This is done by using the information in the AST and building a new tree to describe the outline. In detail, the algorithm works as follows: first, let

$$\begin{aligned} N &= \{\text{AST nodes in a program}\}, \\ S &= \{\text{instances of the Symbol class}\}, \\ T &= \{n \in N \mid \exists s \in S, s.\texttt{getNode()} = n\}, \\ P_n &= \{p \in T \mid p \text{ ancestor to } n\}, \\ & P_n \subset T \subseteq N. \end{aligned} \tag{1}$$

For every node $n \in N$, the tree is traversed bottom-up, and when the first ancestor p with a symbol is found ($p \in P_n$), that node becomes the parent in the outline—that is, n becomes a child of p. The outline depth for n is $d_n = \#(P_n)$, and d_n determines the level of indentation in the hierarchy. This is visualised in Fig. 4 for two different programs in a C-like language:

a) In Fig. 4a), $n = x$ and $P_x = \{\texttt{fac}\}$. The function `fac` is the parent of the variable x, thus letting the variable be a child of the function. This results in one level of indentation ($d_x = 1$).
b) In Fig. 4b), $n = a$ and $P_a = \{\texttt{class}, \texttt{func}\}$. Here, `class` is the parent of `func`, and `func` the parent of `a`. Thus, there are two levels of indentation ($d_a = 2$).

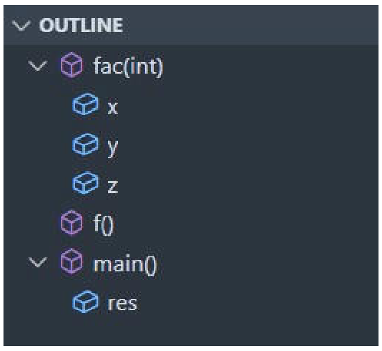


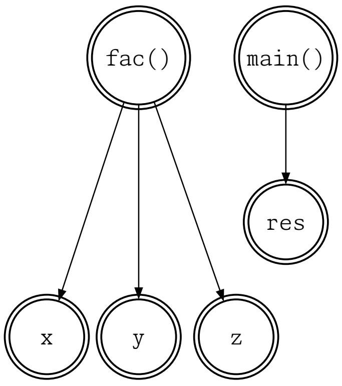
Fig. 3: Hierarchical outline of a program compiled with a JastAdd compiler, showing the functions at the top level and the function variables as children.

B. Code Completion

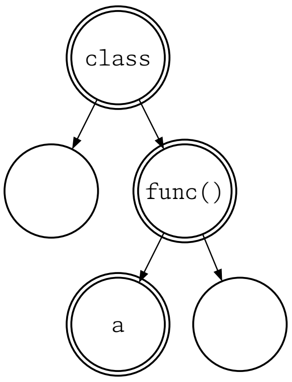
To enable code suggestions to be shown to a developer when writing a program, we added support for defining the code completion items in a compiler, in a similar manner to the Document Outline. An example of Code Completion being used in Visual Studio Code can be seen in Fig. 5. The new special attribute `lsp_completions()` (the full signature can be seen in Fig. 2) can be implemented for a node and must return a Java set `Set<CompletionItem>` containing all names that are available for completion at that node. The node that has precedence is the bottom-most node in the AST that can be found at the current cursor position in the text editor. The class `CompletionItem` is defined in the `interop` library as:

```
class CompletionItem {
  String getName();
  int getKind();
}
```

The attribute could be implemented in different ways by a compiler author depending on the language being created. An appropriate solution for a language that has global scoping rules could be to use a collection attribute [8] at the root node which is then broadcast to all nodes.

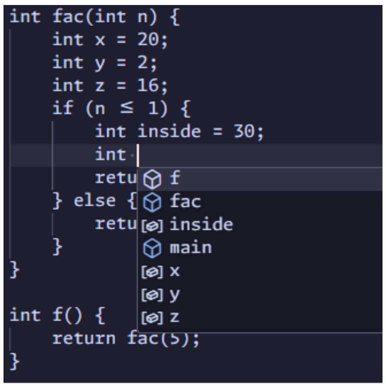


a) One level of outline hierarchy. Corresponds to the outline in Fig. 3. The function `f` is omitted from this figure on purpose.



b) Two levels of outline hierarchy.

Fig. 4: Example of ASTs where the double circled nodes contribute instances of `Symbol` to `lsp_symbols()`. This means that those nodes have a `SymbolKind`, and thus the nodes are elements of T.



```
int fac(int n) {
    int x = 20;
    int y = 2;
    int z = 16;
    if (n ≤ 1) {
        int inside = 30;
        int |
        retu f
    } else { fac
        retu inside
    }        main
}            x
             y
int f() {    z
    return fac(5);
}
```

Fig. 5: Code completions for variables and functions in a C-like language. The values shown in the completion list are correctly scoped, so that the variable `inside` is only visible in the block of the if statement.

When more complex scoping rules are required, a combination of synthesised and inherited attributes could be used instead. This is due to the fact that the Set returned for a node lower in the AST must include the values from higher nodes as well to access them. That is, the broader scope must be unionised with the narrower scope at the lower node.

To trigger completions, a language server can define certain trigger characters. These must however be defined when contact between the server and client is established. Since this cannot be fetched by JAB from the compiler until after the handshake, some trigger characters were defined as a default. If customisation is wanted, the user is recommend to redefine these within the server. Unfortunately, this means that JAB might need some changes itself to work according to a language developer's requirements.

IV. Case Study

During development of new JAB features, continuous internal testing was performed. For this, we mainly used the provided *CalcRAG* project from the repository, and one of our own *SimpliC* compilers from the *EDAN65 Compilers* course at LTH.

However, we preferred to have a more thorough test integration in lieu of relying solely on experimental compilers. Therefore, we chose to perform a case study where JAB was integrated with a larger, feature-complete compiler. In this section, we describe the integration and the achieved results.

A. Introduction

To test if the features of JAB work in a real-world scenario, we decided to integrate it with the ExtendJ compiler. This gave knowledge on whether it would be possible to integrate it with a fully-fledged compiler. Additionally, it gave insights on how the functionality and ease of implementation compared to the small compilers used for continuous testing. More specifically, it demonstrated that LSP features could be added to a Java compiler with little effort.

B. Method

When deciding on which feature-complete compiler to use for this integration, ExtendJ was chosen. It was deemed suitable for the case study due to it being a JastAdd-based compiler for Java [10]. After choosing the compiler, the questions that we sought to answer were:

1) Is it possible to integrate JastAdd Bridge with ExtendJ without considerable effort?
2) If possible, how does this result compare to another Java language server?

The integration of JAB with ExtendJ was created by choosing to implement *Hover*, *Go To Definition*, *Document Outline* and *Code Completion* for Java 8. This version of Java was chosen as it was the latest version that was almost fully supported by ExtendJ [7]. The language server that JAB was compared with is called *Language Support for Java(TM) by Red Hat*, version `1.38.0` available in the VS Code marketplace [11].

To begin with, the ExtendJ repository was forked and cloned. Thereafter, a new `.jrag` file was created with the attributes as defined in Section III. ExtendJ was then compiled and the client settings of JAB in Visual Studio Code were set to point to the ExtendJ compiler. However, the source code of the `interop` library had to be manually copied to ExtendJ as there were some crashing issues, seemingly due to mismatching Java versions when using `interop` as a dependency. Finally, when the implementation was finished, the file was committed to the JAB repository.

C. Results

Integrating JAB and ExtendJ resulted in functioning implementations of `Hover`, `Go To Definition`, `Document Outline` and `Code Completion`, of which the Document Outline can be seen in Fig. 6 and the Code Completion in Fig. 8. This can be compared to the output when using the Red Hat language server, where the document outline can be seen in Fig. 7 and the code completion in Fig. 9.

Minor differences between JAB and the Red Hat language server can be noted. For example, the hierarchy differs in the outline—and unlike JAB, the code completion for the Red Hat language server shows type information and function signatures.

D. Discussion

The integration of JAB and ExtendJ proved to be simpler than initially estimated. Since we were unsure whether it would be

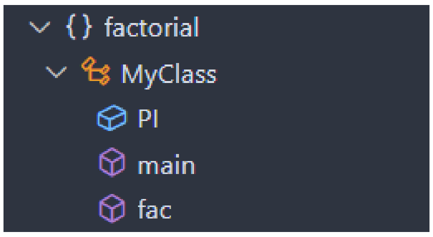


Fig. 6: The document outline for a Java program when using ExtendJ and JastAdd Bridge.

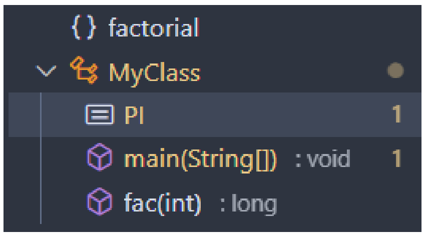
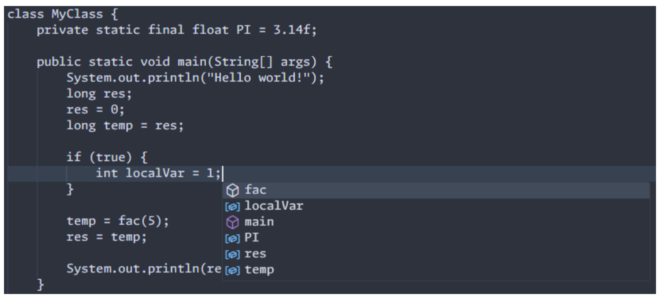


Fig. 7: The document outline for a Java program when using the Red Hat language server.

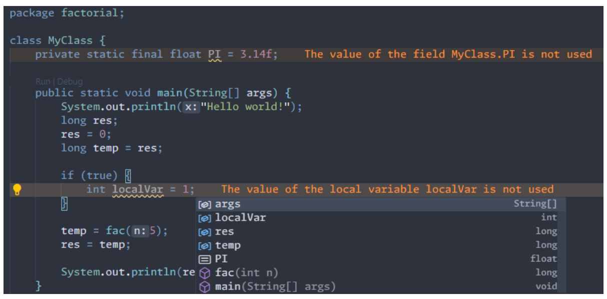


```
class MyClass {
    private static final float PI = 3.14f;

    public static void main(String[] args) {
        System.out.println("Hello world!");
        long res;
        res = 0;
        long temp = res;

        if (true) {
            int localVar = 1;
        }                     fac
                              localVar
        temp = fac(5);        main
        res = temp;           PI
                              res
        System.out.println(re temp
    }
```

Fig. 8: Code completion list for a Java program when using ExtendJ and JastAdd Bridge.



```
package factorial;

class MyClass {
    private static final float PI = 3.14f;    The value of the field MyClass.PI is not used

    Run | Debug
    public static void main(String[] args) {
        System.out.println(x: "Hello world!");
        long res;
        res = 0;
        long temp = res;

        if (true) {
            int localVar = 1;    The value of the local variable localVar is not used
        }                    args                        String[]
                             localVar                    int
        temp = fac(n: 5);    res                         long
        res = temp;          temp                        long
                             PI                          float
        System.out.println(re fac(int n)                 long
    }                        main(String[] args)         void
```

Fig. 9: Code completion list for a Java program when using the Red Hat language server.

feasible to integrate it or not considering the time constraints, we were pleased to see the results working as we expected. Overall, we were generally satisfied with both the results and by the fact that they did not differ by a significant amount compared to Red Hat's language server. The integration was achieved with a `.jrag` containing about 100 lines of code (LOC), and a part of it is displayed in Fig. 10.

Worth noting is that we simplified the implementation of code completion slightly by not checking whether a variable or a field is a constant, which is why the icons for `PI` in Fig. 8 and Fig. 9 differ. Also, we did not include the parameter `args` in the completion list, however adding it would not be a significant adjustment.

After performing the first iteration of our integration, we noticed that its appearance was similar to the Red Hat language server's. Judging by the small differences, it seemed like only minor modifications had to be made to match that appearance. Thus, we aimed to update the code to achieve a more similar result. Consequently, we were pleased to see that we could easily achieve comparable functionality and appearance to an established language server. One thing we considered remarkable was achieving this with a minimal working solution. Furthermore, implementing basic scoping rules was also easier than expected, as it was similar to the implementation for SimpliC.

A noteworthy detail in Fig. 6 is that the document outline for JastAdd Bridge shows the class nested under the package name, which is not the case for Red Hat's server in Fig. 7. Removing the nesting to match Red Hat's implementation is impossible due to the automatically hierarchical document outline that we described in Section III–A. This could be considered undesirable behaviour and a limitation in cases where a lack of nesting is expected.

Unfortunately, the bugs that affected both the CalcRAG and the SimpliC compiler were also apparent when using ExtendJ, in addition to other issues. Ideally, we would like to fix these issues if more time was available. For example, the issue with `interop` causing crashes resulted in a less adequate solution. Due to the aforementioned time constraint in addition to not knowing the root cause, it was decided that we copy the source code of `interop` to ExtendJ in order to get a minimal working solution.

Initially, a lack of fault tolerant parsing caused issues, but it was resolved after completion of the case study. With our implementation in place, if the ExtendJ compiler failed its parsing, it immediately crashed the JAB server. This required a restart of VS Code. This was a critical issue since it meant that no unparsable programs could be written, significantly reducing the usefulness of JAB. This issue, in combination with enabling auto-saving in VS Code per our own recommendation, made it entirely unusable. At first, it resulted in the desired integration barely being achieved. The issue was resolved by introducing the function `CodeProber_parse` to ExtendJ's main program.

To summarise, the case study resulted in a working demonstration of the integration between JastAdd Bridge and Extendj in around 100 LOC—notably with simple scoping rules that could be extended to cover the full Java scoping rules without considerable effort. Furthermore, imitation of the behaviour of Red Hat's language server was achieved with minor adjustments. Lastly, not all JAB features were integrated due to time constraints. Due to issues with an unknown root cause, the `interop` library had to be manually copied into ExtendJ.

V. User study

JastAdd Bridge is a tool that aims to aid developers when programming. Introducing functionality while disregarding usability and user experience is not enough for it to be appreciated by users [12, pp. 520–521]. Providing a well-designed system is therefore an important factor for users to appreciate it. Therefore, evaluating the users' interaction with the system is an appropriate way to gain the required understanding of their opinions.

A. Introduction

One way of conducting evaluations for technological products is to perform *In-the-Wild Studies* [12, p. 530]. This is a type of evaluation that is performed in regular environments outside of testing labs. Compared to other more controlled types of evaluation, the evaluators let the participants use the system with minimal involvement. This gives a realistic indication of how the system is used and what opinions the users have of it. One method that can be used to perform an In-the-Wild Study is to conduct interviews.

```
aspect LSPOutline {
  coll Set<LSPSymbol> Program.lsp_symbols() [new HashSet()]
   with add root Program;

  ClassDecl contributes LSPSymbol.of(
   getID(), SymbolKind.CLASS, lsp_range(), this
  )
  to Program.lsp_symbols() for program();

  MethodDecl contributes LSPSymbol.of(
   getID(), SymbolKind.METHOD, lsp_range(), this
  )
  to Program.lsp_symbols() for program();

  // ...
```

Fig. 10: A part of the `.jrag` file for ExtendJ.

B. Method

During the development of our additions to JAB, the features were designed with usability and user experience in mind. However, to evaluate the real level of these metrics, In-the-Wild interviews were carried out. Specifically, semi-structured interviews were carried out, meaning that additional questions could be asked, despite the use of prepared questions [12, p. 286].

Three students that had previous experience of compiler development in JastAdd volunteered to participate in the study. The criteria for participation was based on whether they had taken the course *EDAN65 Compilers*, which is taught at Lund University.

Before conducting the interviews, the questions found in Appendix A were prepared. Furthermore, a consent form that the participants were required to sign was prepared. For the presentation of the results, we will use the aliases A, B and C for the participants.

C. Results

Out of the three participants, only interviewee C knew of LSP, although all participants regularly use features typically provided by language servers. All three used VS Code as their primary editor, with interviewee A also using Vim.

The terminology for certain features, such as “outline”, was not intuitive for the participants, and none of them recognised the term initially. It became clear after reading our documentation or seeing the feature in VS Code. Both interviewees A and B found the outline potentially helpful for navigating larger projects, but interviewee C preferred using derivative features like *Show All Symbols* or *Go to Symbol* in the command palette instead. Worth noting is that Go to Symbol utilises the outline within the current file, whereas Show All Symbols goes through all symbols in the workspace. This means that JAB does implement Go to Symbol, but it would require multi-file support for the full feature.

Regarding code completion, all participants were familiar with it and had used the feature, though interviewee B initially confused it with AI-assisted code generation (e.g. GitHub Copilot). All participants agreed that `Code Completion` was a valuable tool that they frequently used during development.

In regards to further improving JAB, some changes were suggested. Interviewee A expressed that a more complex code completion is desired, exemplified with their preferred Java language server suggesting to replace the text `main` to a full function signature, as well as placing the cursor within the body of the function. As of right now, JAB only supports code completion inserts, not replacements, and the cursor cannot be placed arbitrarily. This might be mitigated by utilising diagnostics combined with a quick-fix edit, but it is not tested. Show all Symbols in VS Code is dependent on multi-file support, but participants otherwise struggled to come up with features not yet implemented in JAB that they regularly use.

a) *Implementation experience:*

In this section we present our observations from the interviews, as well as feedback given by the participants. At the time of the interviews, there were no clear examples on, or pictures of specific features in the documentation.

Document Outline: Implementing the Document Outline feature presented varying levels of difficulty among participants. However, all participants managed to complete the task.

Interviewee A communicated that they had forgotten some JastAdd concepts since taking the Compilers course and this likely caused the initial difficulty. They also expressed that there was a lot of information to process when reading the documentation. It was overwhelming to understand which parts were actually relevant to the task in a short amount of time.

Interviewee B first wanted to understand what the feature was, and read only the outline section in the documentation. They understood the task after seeing an example of an outline shown in VS Code, and could easily add a symbol to the outline using JastAdd after that. They mainly used examples found in the repository to aid with JastAdd syntax to complete the task.

Interviewee C wanted to read a minimal amount of documentation, and thus skimmed through some of the paragraphs in the outline section of the documentation. They initially did not understand the role of JAB in simplifying protocol handling but accepted the task after an explanation of the separation of semantics and protocol handling using the specially named attributes. They were prompted to read some specific paragraphs in the documentation, and thus found examples in the repository. After this, they managed to complete the task.

All interviewees found the process manageable after some guidance on getting started, and relied heavily on examples within the provided repository. Common feedback included that the `README` could benefit from more examples showing clear steps for adding AST nodes to the outline, especially for participants less familiar with JastAdd or compiler design. Additionally, explaining key concepts like SymbolKind and the connection to the LSP definition in more detail would help reduce confusion.

Code Completion: For the code completion task, all interviewees found the implementation easier after completing the outline task, as the two features shared similarities.

However, interviewee C thought the trivial implementation of a global suggestion incomplete and in its current state, unnecessary. They cited issues with a lack of fault-tolerant parsing and unclear triggers for suggestions. This is due to the `CalcRAG` compiler itself, and as such, we discussed ways this could be better implemented. We found that it would require changes to the language and parsing, and using inherited attributes for code completion to achieve the result they expected, but that it is ultimately out of scope for JAB to try to handle it automatically.

In general, participants wished for more detailed examples in the `README`, showing how code completion integrates with typical use cases, such as variable declarations and scoping. While they appreciated the feature itself, interviewee B and C highlighted that implementation guided solely by our documentation is difficult due to `CompletionItemKind` and its connection to LSP not being explained properly in the same manner as `SymbolKind`. They also recognised that code completion would be harder to implement for non-trivial cases i.e. with scoping rules.

D. Discussion

Although this was a small and homogenous group, their approaches to the tasks differed significantly. Interviewee A was a thorough reader, while interviewee C aimed to avoid reading unnecessarily. Interviewee B fell somewhere in between, as they wanted to understand the reason for certain implementation details. While we cannot draw any definitive conclusions due to the small number of participants, we still gained valuable insights from their different approaches. They also offered helpful suggestions on improving the documentation, which has since been updated with clearer examples.

We had anticipated that a user of Vim would utilise the outline feature more, thus motivating the implementation of it. This was not the case, but the feature was still deemed useful by all participants due to it providing an overview of the current file and position, and fast navigation.

The results of the study seems to indicate that the most important LSP functionalities are implemented, but that it needs to be improved. Multi-file support is needed for full support of current features, and changes to Code Completion are desired. Code Completion supporting inserts only was a choice we made during development as we thought it better to simplify the integration between a compiler and JAB. However, this study shows that there is an interest in more flexibility.

In general, participants indicated that they appreciated JAB, and could see real use cases for it.

VI. Related Work

Most relevant to this paper is the previous work by Hemberg and Hardt [5], creating the initial version of JastAdd Bridge. As pointed out in their work, a similar solution to JAB is MagpieBridge [13].

Additionally, CodeProber which is a live exploration tool for program analysis results [14], is used as a dependency in JAB. CodeProber is either used to locate AST nodes, extract position information from an AST node, or to get syntax errors for diagnostics [5].

Earlier participants of the EDAN70 course have made their own LSP servers specifically for ExtendJ, which could be compared with the implementation in our case study. One project implemented a server and extensions for two different editors, capable of error handling [15]. Another project implemented a Language Server where ExtendJ was used as the backend, in a similar manner as to how JAB assumes that the compiler calculates the semantics [16].

VII. Conclusion

We continued working on JastAdd Bridge, a tool that simplifies implementing language server functionality for languages defined with the meta-compilation system JastAdd. JAB handles communication through the language server protocol, which enables use of all editors which implement it.

Our main focus was expanding supported functionality and evaluating the project, mainly focusing the evaluation on our additions and further work. Additions to functionality enable users to:

- create an automatically hierarchical outline for their language.
- easily define which AST nodes should be included in the outline.
- customise the labels that are shown in the outline.
- show general code completion suggestions for certain nodes.
- implement scoped code completions.

To do this, a JastAdd compiler should handle the semantics of its own language, and expose the results to JAB through certain JastAdd attributes. The attributes must match our specification for the integration to work.

A. *Further work*

There are many alternatives to improve JAB, where several approaches were mentioned in the initial paper [5]. Below, we present some alternatives in order of descending perceived importance.

Documentation: As made clear during our user study, the documentation was not easy to understand for an uninitiated user. Even though we tried to remedy this after the study, we suspect much more should be done. We were planning on dividing the documentation into smaller, more easily readable parts, and expand the usage examples. Ultimately, we did not have time to do so, but we believe that it could be an appreciated improvement.

Responsiveness: This is something we did not work on, and as such, the introducing paper explains the state quite well [5]. We have however added the recommendation of setting VS Code to autosave the current file every second, but this does not resolve the underlying issue. Additionally, when using a compiler that does not have a fault-tolerant parser, it can possibly cause crashes.

Multi-File support: According to both our small user study and internal evaluation, the next feature to be implemented should be multi-file support. It is necessary for the full utilisation of `Go to Definition`, or outline through `Show all Symbols`. Not only that, there are few real life applications in which all source code is contained within one file.

ExtendJ integration: It would be interesting to continue improving the integration with ExtendJ, both to be able to test the limits of JAB, but also to enable extended use of the alternative compiler in the future.

Other LSP functionality: Here, both improvements and new functionality could be implemented. For example, enabling details to be added to a `DocumentSymbol` in the outline to show the type of a function would improve user experience. The same applies to `CompletionItem`, which contains both the field `detail` for general info like type or symbol information, and `labelDetails` containing `CompletionItemLabelDetails` for function signatures or type annotations and descriptions.

Otherwise, as indicated by the user study, the possibility of not only inserting completions, but replacing code could be a good addition.

Furthermore, we found a study analysing 30 different language servers and lists each server's implemented LSP features [17]. This could be utilised to identify relevant functionality to add to JAB.

Expansion to other editors: As JAB implements LSP, it has the potential to be used with many modern editors. To support this, each editor needs to have client-side specific code, but the server could remain as is. The client is still assumed to support all functionality of which the server supports, which is not always true, and should be fixed to reduce the number of unnecessary messages. This issue should be resolved in the server, as contact is being established. Furthermore, we have not implemented another communication channel, and as such, only communication through a WebSocket is supported.

Custom trigger characters: A possible improvement is adding the ability to customise the trigger characters used for Code Completion in a more user-friendly manner. Currently, compiler developers need to modify the source code of the JAB server to modify it. Although the required changes are minimal, being able to specify custom trigger characters from the client or from within the compiler would improve the user experience.

Acknowledgements

We would like to thank Johannes Hardt and Dag Hemberg for laying the groundwork of JastAdd Bridge, and Anton Risberg Alaküla for his enthusiastic supervision and help during this project. We also thank the anonymous participants of our user study for their time and feedback.

Appendix A

A. *Questions for User Evaluation*

The following questions were asked at the interviews in the user evaluation:

a) *General questions:*

1) Do you know what the LSP (Language Server Protocol) is? Do you normally use a Language Server or something similar when developing programs?
2) What IDE do you normally use when you are programming?

b) *Questions regarding document outline:*

1) Do you know what the outline in your editor is? Is it something that you usually use when you are programming in general?
2) We have prepared a compiler for the language *CalcRAG* with the corresponding aspect files. We would like you to read the `README` of JastAdd Bridge and try to add an AST node of your choice to the outline.
3) [When the interviewee is done]:
   - How difficult did you consider it was to implement the outline?
   - Do you think it is a useful function?

c) *Questions regarding code completion:*

1) Do you know what code completion is? Is it something that you usually use when you are programming in general?
2) We would like you to read the `README` again and try to make a AST node of your choice work with the code completion.
3) [When the interviewee is done]:
   - How difficult did you consider it was to implement code completion?
   - Do you think it is a useful function?

d) *Closing questions:*

1) Did you think it was easy to follow the `README` to understand what you had to do?
2) Is there anything you felt needed a better explanation, or if anything is missing in the `README`?
3) When you read about the features listed in the `README`, were there any other feature that you would like to see implemented in JastAdd Bridge?

References

[1] J. Kjær Rask, F. Palludan Madsen, N. Battle, H. Daniel Macedo, and P. Gorm Larsen, “The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions,” *Electronic Proceedings in Theoretical Computer Science*, vol. 338, pp. 3–18, Aug. 2021, doi: 10.4204/eptcs.338.3.

[2] G. Hedin and E. Magnusson, “JastAdd — an aspect-oriented compiler construction system,” *Science of Computer Programming*, vol. 47, no. 1, pp. 37–58, 2003, doi: 10.1016/S0167-6423(02)00109-0.

[3] G. Hedin, “Reference attributed grammars,” *Informatica*, vol. 24, no. 3, pp. 301–317, 2000.

[4] D. E. Knuth, “Semantics of context-free languages,” *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, 1968, doi: 10.1007/bf01692511.

[5] D. Hemberg and J. Hardt, “JastAdd Bridge: Interfacing reference attribute grammars with editor tooling,” *Lund University*, 2023, [Online]. Available: https://fileadmin.cs.lth.se/cs/Education/edan70/CompilerProjects/2023/Reports/hardt-hemberg.pdf

[6] “Eclipse-Lsp4j/Lsp4j: A Java Implementation of the Language Server Protocol Intended to Be Consumed by Tools and Language Servers Implemented in Java..” Accessed: Dec. 05, 2024. [Online]. Available: https://github.com/eclipse-lsp4j/lsp4j

[7] “ExtendJ - The JastAdd Extensible Java Compiler.” Accessed: Dec. 05, 2024. [Online]. Available: http://extendj.org/

[8] E. Magnusson, T. Ekman, and G. Hedin, “Extending Attribute Grammars with Collection Attributes — Evaluation and Applications,” *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, pp. 69–80, 2007, doi: 10.1109/SCAM.2007.13.

[9] Microsoft, “Language Server Protocol Specification — 3.17.” Accessed: Nov. 20, 2024. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/

[10] J. Öqvist, “ExtendJ: Extensible Java Compiler,” in *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, Nice, France: ACM, Apr. 2018, pp. 234–235. doi: 10.1145/3191697.3213798.

[11] “Language Support for Java(TM) by Red Hat.” Accessed: Jan. 08, 2025. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=redhat.java

[12] H. Sharp, Y. Rogers, and J. Preece, *Interaction Design: Beyond Human-Computer Interaction*, 6th ed. Hoboken: John Wiley & Sons, Inc, 2023.

[13] L. Luo, J. Dolby, and E. Bodden, “MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper),” *LIPIcs, Volume 134, ECOOP 2019*, vol. 134, pp. 1–25, 2019, doi: 10.4230/LIPIcs.ECOOP.2019.21.

[14] A. Risberg Alaküla, G. Hedin, N. Fors, and A. Pop, “Property Probes: Source Code Based Exploration of Program Analysis Results,” in *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, Auckland, New Zealand: ACM, Nov. 2022, pp. 148–160. doi: 10.1145/3567512.3567525.

[15] F. Siemund and D. Tovesson, “Language Server Protocol for ExtendJ,” Dec. 2018, [Online]. Available: https://fileadmin.cs.lth.se/cs/Education/edan70/CompilerProjects/2018/Reports/SiemundTovesson.pdf

[16] J. Ericson, “Language Server Protocol for ExtendJ,” Feb. 2019, [Online]. Available: https://fileadmin.cs.lth.se/cs/Education/edan70/CompilerProjects/2018/Reports/Ericson.pdf

[17] D. Barros, S. Peldszus, W. K. G. Assunção, and T. Berger, “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*, in MODELS '22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 232–243. doi: 10.1145/3550355.3552452.