Developing a Minimal Language Server for the Frege Programming Language: an Experience Report

Thibault Gagnaux
Supervisor: Prof. Dierk König
University of Applied Sciences and Arts Northwestern Switzerland
Institute of Mobile and Distributed Systems
Windisch, Switzerland
tgagnaux@gmail.com

***Abstract*—Language servers provide features such as code completion and documentation on hover to text editors and integrated developer environments. These features help developers to write software more efficiently but take a significant time to build. The language server protocol addresses this problem by allowing to write a language server once and reuse it with many popular text editors. In this experience report, I present the steps needed to develop a minimal language server for the Frege programming language communicating with a Visual Studio Code plugin over the language server protocol. I start with Microsoft's example language server and gradually extend it. Firstly, I port the example server from Typescript to Java and secondly, I integrate the Frege compiler to show compiler errors and type signatures on hover. This experience report should help others to estimate the work of developing a language server and how to approach this task.**

***Index Terms*—language server, ide, frege**

I. Introduction

Integrated development environments (IDEs) help developers to write software faster. IDEs are enhanced text editors with extra features to basic text editing. Among others, they show warnings and errors after every source code change, provide additional information while hovering over a code expression and make suggestions to autocomplete your code. These features are the most often used actions by developers besides the common copy, paste, delete and save commands [1].

The abovementioned features are often powered by a language server, a standalone program which analyses the source code and returns the result to the IDE. Developing a language server is tightly coupled to one programming language and one IDE. Jetbrains, a company specialised in creating IDEs, currently offers more than 10 IDEs, each one targeted for a different programming language. As a result, m programming languages and n IDEs require $m \times n$ language servers as shown in Table I.

The integration of multiple IDEs is thus a considerable effort because the development of a single language server is already a time-consuming task. Microsoft has identified this problem and standardised the communication between a client, e.g. an IDE and a language server, with the language server protocol (LSP) [2]. As a result, every language server implementing the LSP can be used with any IDE implementing the LSP. This reduces the $m \times n$ matrix problem to m reusable language servers + n reusable IDE plugins as depicted in Table II.

TABLE I
THE m PROGRAMMING LANGUAGE AND n IDE MATRIX PROBLEM RESULTS IN $m \times n$ LANGUAGE SERVERS.

	Visual Studio Code	Eclipse	$\cdots$	IDE n
Java	Language Server 1	Language Server 2	$\cdots$	Language Server n
Python	Language Server $n+1$	Language Server $n+2$	$\cdots$	Language Server $n \times 2$
$\vdots$	$\ddots$	$\ddots$	$\ddots$	$\vdots$
Language m	$\cdots$	$\cdots$	$\cdots$	Language Server $m \times n$

TABLE II
IF BOTH THE LANGUAGE SERVER AND THE IDE SUPPORT THE LANGUAGE SERVER PROTOCOL (LSP) THEN m PROGRAMMING LANGUAGES AND n IDES NEED m LANGUAGE SERVERS + n IDE PLUGINS.

	Visual Studio Code	Eclipse	$\cdots$	IDE n
Java	Language Server 1 + Plugin 1	Language Server 1 + Plugin 2	$\cdots$	Language Server 1 + Plugin n
Python	Language Server 2 + Plugin 1	Language Server 2 + Plugin 2	$\cdots$	Language Server 2 + Plugin n
$\vdots$	$\ddots$	$\ddots$	$\ddots$	$\vdots$
Language m	$\cdots$	$\cdots$	$\cdots$	Language Server m + Plugin n

However, many IDE plugins still contain server specific configurations in practice, resulting in more than n needed plugins. Nevertheless it is still beneficial that the feature-powering language server is reusable because it takes much more time to develop the language server than the IDE plugin.

In this experience report, I present the steps needed to develop a minimal language server [3] for the programming language Frege [4] and its corresponding plugin [5] for the Visual Studio Code editor using the LSP. Frege is a purely functional programming language, which compiles to Java [6] and can therefore be used with every Java project. Given its niche programming language status, Frege is currently still missing a language server implementing the LSP. On the plugin side, I chose Visual Studio Code because according to the StackOverflow survey 2021 [7] it is the most used IDE across all developers.

The rest of the paper is structured as follows: Section II shows which language tools already exist for Frege. Section III gives a high-level overview of the Frege language server and its components. Section IV and Section V explain the language

server protocol and Frege Java interoperability boundaries and demonstrate how I gradually extended the language server. In Section VI the features and limitations of the final Frege language server are discussed. I conclude with Section VII.

II. Related Work

There are already two existing IDE dependent Frege language plugins: Firstly, Ingo Wechsung, the creator of Frege, has developed a plugin for the Eclipse IDE [8] called fregIDE [9]. It depends on the IDE Metatooling Platform [10], which aims to improve the development of language features in Eclipse. Secondly, for IntelliJ IDEA, an IDE by Jetbrains, there is active development on a language plugin [11], which is based on IntelliJ's Program Structure Interface (PSI). Furthermore, the Frege read-eval-print-loop (REPL) [12] provides an interactive Frege environment, which allows to execute Frege code either in the terminal or in the browser[1]. It is of special interest because it directly uses the Frege compiler to power similar features to a language server.

The three abovementioned language tools share the same limitation: they only work within their intended IDE and are thus not reusable. Developing a Frege language server conforming to the language server protocol removes that limitation and makes the Frege language features available to every editor supporting the language server protocol such as Visual Studio Code, Eclipse, Neovim, Emacs, Atom, THEIA, Sublime Text and many more[2].

III. Concept

I want to illustrate the Frege language server's architecture with the help of the hover feature. Given the simple Frege file `Frobnicate.fr` in Listing 1.

```
1  module Frobnicate where
2
3  frob a = (a * a, "Frege rocks")
```

Listing 1: The `frob` function defined in the `Frobnicate.fr` file.

Whenever I hover over the function `frob` on line 3 in Visual Studio Code, I wish to see its function signature `Num a => a -> (a, String)` in an overlay as shown in Figure 1.

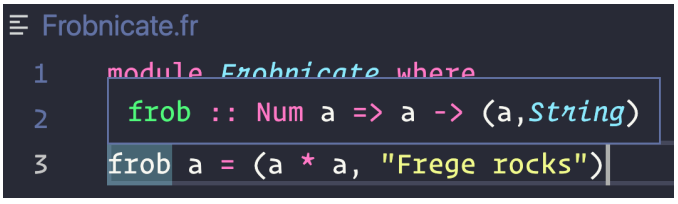

Fig. 1. Hovering over the frege function `frob` in Visual Studio Code shows its type signature `frob :: Num a => a -> (a, String)`.

[1]See http://try.fregelang.org

[2]See https://microsoft.github.io/language-server-protocol/implementors/tools/

Hence the hover feature needs at least the following three components as depicted in Figure 2:

The Visual Studio Code Plugin making the hover request.
The Frege Language Server receiving the request.
The Frege Compiler evaluating the file and returning the type signature.

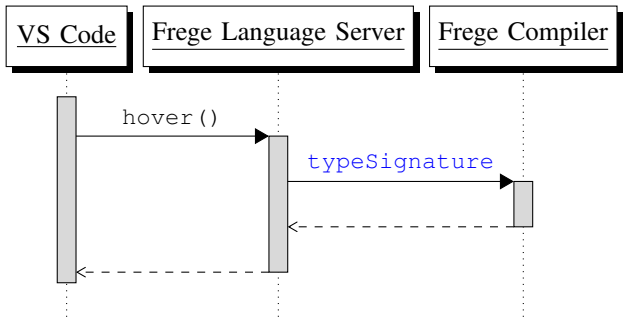

Fig. 2. Hover sequence diagram: Hovering over a function name in Visual Studio Code (VS Code) calls the Frege Language Server, which calls the Frege Compiler to infer the type signature and returns it to Visual Studio Code.

The abovementioned three components come with two boundary integration problems: the Visual Studio Code Plugin and the Frege language server communicate over the language server protocol and the Frege language server needs to interact with Frege code since the Frege compiler is written in Frege. I solved these two integration problems iteratively with multiple proof of concepts and will present them in the next two Sections.

IV. Language Server Protocol Boundary

The language server protocol uses a remote procedure call JSON (JSON-RPC) message format exchanged between a client and a server. The protocol is divided into a header and content part and specifies three types of messages:

A request message which must be answered with a response message.
A response message for the request message.
A notification message which expects no response message and can thus be treated like an event.

With these three types of messages the language server protocol defines the reusable set of features. Among others, these reusable features include general initialisation, window, text synchronisation, diagnostics and multiple language feature messages. For example, hovering over the `frob` function in Listing 1 in Visual Studio Code sends the message shown in Listing 2 to the Frege Language server. The full list of features are documented on the specifications website[3].

[3]See https://microsoft.github.io/language-server-protocol/specifications/specification-current/

```
Content-Length: 216\r\n
\r\n
"jsonrpc": "2.0",
"id": 1,
"method": "textDocument/hover",
"params": {
    "textDocument": {
        "uri":
        ↪  "file:///Users/.../Frobnicate.fr"
    },
    "position": {
        "line": 2,
        "character": 2
    }
}
```

Listing 2: The `frob` function defined in the **`Frobnicate`**`.fr` file.

A. First Proof of Concept

As a first step to understand the language server protocol, I followed Microsoft's tutorial [13] on how to write a basic language server implementing the language server protocol. The basic language server is written in Typescript and provides diagnostics and autocompletion for `.txt` files as shown in Figure 3.

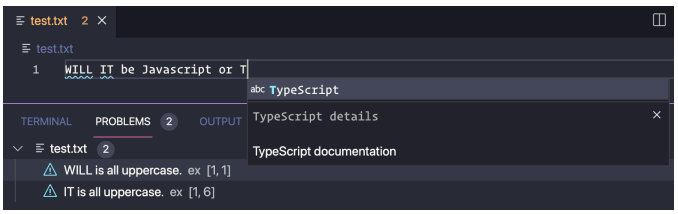

Fig. 3. The basic language server provides two features: Firstly, it publishes diagnostics for uppercase words of length 2 and more. Secondly, if you type the character *j* or *t* it provides the autocompletion *JavaScript* or *TypeScript* respectively.

Implementing the language server protocol from scratch takes a considerable effort. Besides, most messages are exchanged asynchronously, which adds complexity. As an alternative, there are generic library implementations in different programming languages available. They abstract the JSON-RPC messages into data structures and map the asynchronous message exchange to the programming language's asynchronous computing model. For example, the basic language server uses the *VSCode Language Server - Node* [14] library. The autocompletion feature is then abstracted to the code shown in Listing 3.

```
connection.onCompletion(
(_textPos: TextDocumentPositionParams):
↪  CompletionItem[] => {
  return [
      {
        label: 'TypeScript',
        kind: CompletionItemKind.Text,
        data: 1
          },
      {
        label: 'JavaScript',
        kind: CompletionItemKind.Text,
        data: 2
          }
    ];
  }
);
```

Listing 3: The autocompletion feature using the *VSCode Language Server - Node* [14] library. It abstracts the language server JSON-RPC message to Typescript types, such as `CompletionItem[]` and `CompletionItemKind.Text`.

B. Second Proof of Concept

Frege compiles to Java. As a result, Java code can call Frege code and Frege code can use Java code. Consequently, I had to make a choice whether to write the Frege Language Server in Java or Frege. Choosing Frege makes the integration with the Frege compiler easier but has the considerable disadvantage that the JSON-RPC message protocol needs to be implemented from scratch. On the other hand, writing the Frege Language Server in Java, gives access to the well-established LSP4J [15] library, which provides a Java binding for the language server protocol.

As a second step, I thus replaced the Typescript basic language server with a Java language server [16] using LSP4J. The autocompletion feature from Listing 3 can be easily mapped using LSP4J as shown in Listing 4.

```
@Override
public
↪  CompletableFuture<Either<List<CompletionItem>,
↪  CompletionList>> completion(CompletionParams
↪  position) {
  List<CompletionItem> completionItems =
  ↪  Arrays.asList(
    createTextCompletionItem("TypeScript", 1),
    createTextCompletionItem("JavaScript", 2)
  );
  return CompletableFuture.completedFuture(
    Either.forLeft(completionItems));
}
```

Listing 4: The autocompletion feature using the LSP4J [15] Java library. LSP4J abstracts the language server JSON-RPC message to Java types, such as `CompletionItem` and `CompletionParams` and uses the `CompletableFuture` class for asynchronous computations.

The main change is that LSP4J uses the `CompletableFuture` class to account for the asynchronous compute model. Hence, the most time-consuming changes were not porting the autocompletion and diagnostic features to Java but starting the server. While the Node library also includes a node server out of the box where the client

and the server connect through interprocess communication (IPC), LSP4J does not. Instead, the client starts the server by calling an executable and they communicate over standard input and output. Maven [17] and Gradle [18] are by far the most popular Java build systems to create an executable Java application according to the Jetbrains state of developer ecosystem report 2021 [19]. I chose Gradle to create operating system specific start scripts. The Visual Studio Code client then starts a new process which executes the start scripts to run the basic language server as shown in Listing 5.

```
let javaServerOptions: ServerOptions = {
  run: { command: "sh", args: [
  ↪  javaLanguageServerStartScriptPath ] },
  debug: {
  command: "sh", args: [
  ↪  javaLanguageServerStartScriptPath ]
  }
}

client = new LanguageClient(
  'basicLanguageServer',
  'Basic Language Server',
  javaServerOptions,
  clientOptions
);
client.start();
```

Listing 5: Starting a basic Java language server from the Visual Studio Code client. The `javaLanguageServerStartScript` is generated with the Gradle task *startScripts* [20].

Debugging the basic Java server requires some tweaking as well. Two steps are needed: Firstly, the Java process must be started in debug mode on an exposed TCP/IP port and secondly a debug attach configuration with a matching port must be specified in Visual Studio Code's `launch.json` file as shown in Listing 6.

```
{
  "type": "extensionHost",
  "request": "launch",
  "name": "Launch Client",
  "runtimeExecutable": "${execPath}",
  "args": [
  ↪  "--extensionDevelopmentPath=${workspaceRoot}"
  ↪  ],
  "env": {
    "JAVA_OPTS":
    ↪  "-agentlib:jdwp=transport=dt_socket,server=
    ↪  y,suspend=n,address=localhost:6008,quiet=y",
  },
  "outFiles": [
  ↪  "${workspaceRoot}/client/out/**/*.js" ],
  "preLaunchTask": {
    "type": "npm",
    "script": "watch"
  }
},
{
  "type": "java",
  "name": "Attach to Java Server",
  "request": "attach",
  "hostName": "localhost",
  "port": 6008
}
```

Listing 6: The two needed launch tasks to debug the basic Java language server: The *Launch Client* task starts the Java process in debug mode on port 6008. Please take special note that the option `quiet=y` is necessary. The *Attach to Java Server* task then connects the debugger to the listening Java process on port 6008.

V. Frege Java Interoperability Boundary

Given a working basic Java language server, only one boundary remains to build a Frege language server: Calling the Frege compiler, which means calling Frege Code from Java. Listing 7 shows the generated Java code of the `frob a = (a * a, "Frege rocks")` function from the Frege file **`Frobnicate`**`.fr` depicted in Listing 1.

```
// ...
// Java and Frege imports removed for brevity
final public class Frobnicate {

  final public static <α> PreludeBase.TTuple2<α,
  ↪  String/* <Character> */> frob(final
  ↪  PreludeBase.CNum<α> ctx$1,
      final Lazy<α> arg$1) {
    return PreludeBase.TTuple2.<α, String/*
    ↪  <Character> */>mk(
        Thunk.<α>shared((Lazy<α>) (() ->
        ↪  ctx$1.f$star(arg$1, arg$1))),
        Thunk.<String/* <Character> */>lazy("Frege
        ↪  rocks"));
  }
}
```

Listing 7: The generated Java code by the Frege compiler from the **`Frobnicate`**`.fr` shown in Listing 1.

The generated Java code from Frege helps a little on how we can call the `frob` function. The first argument of the Java `frob` function has the type `PreludeBase.CNum`. This corresponds to the Frege type class **`Num`** and therefore the second argument needs to be a value of the type class **`Num`**. But the type of that value is `Lazy`. Fortunately, the underlying topic of Thunks and Boxes is well documented on the Frege wiki, which provides

the solution: "To get a `Lazy<R>` of a value with type `R`, pass the value to `Thunk.<R>lazy`" [21]. Listing 8 shows a passing test case of how we can call the Frege `frob` function in Java.

```
@Test
void can_call_frobnicate() {
  int expectedNumber = 4;
  String expectedString = "Frege rocks";
  TTuple2<Integer, String> actual =
  ↪  Frobnicate.frob(PreludeBase.INum_Int.it,
  ↪  Thunk.lazy(2)).call();
  assertEquals(expectedNumber, actual.mem1.call());
  assertEquals(expectedString, actual.mem2.call());
}
```

Listing 8: A Java test case which shows how to call the Frege `frob a = (a, "Frege rocks")` function introduced in Listing 1 from Java.

With that knowledge, I extended my basic Java language server to call a Frege function: Whenever the character *f* is typed, the autocompletion feature now provides the "Frege rocks" suggestion.

However, calling Frege from Java has the constraint that the Frege code must first be compiled to Java. The automation of this step means a significant change to the Gradle build tool process because the *compileFrege* task becomes the new root of the Gradle Java plugin's task dependency graph shown in Figure 4.

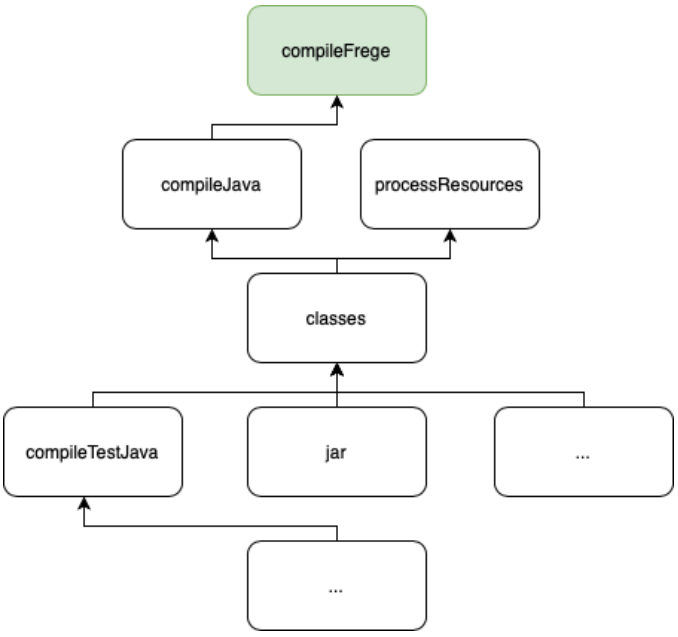

Fig. 4. The *compileFrege* task marks the new root dependency in the Gradle Java plugin's task dependency graph. A simplified version is shown here. See [22] for all tasks and their dependencies.

Therefore, I built a custom Frege gradle plugin [23], which adds the needed *compileFrege* Gradle task. *The compileFrege* downloads the specified version of the Frege compiler and compiles all `*.fr` files to the `build/classes/frege` output directory.

With all that in place, I was ready to call the Frege compiler to get the type signature of the `frob` function, which I will describe in the following Section.

A. The Frege Compiler

In order to call the Frege compiler I reused the logic from the Frege REPL project [12]. The Frege compiler is accessed through the **`FregeRepl`** and the **`FregeInterpreter`** modules as depicted in Figure 5.

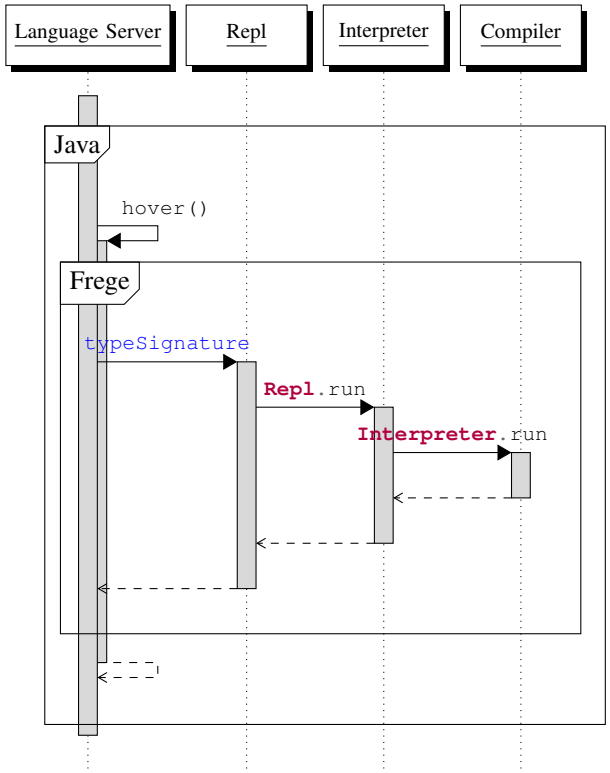

Fig. 5. The Frege language server accesses the Frege compiler through the Frege Repl and Frege Interpreter modules whenever Visual Studio Code sends a `hover()` request as shown in Figure 2.

Monad transformers [24] are the driving concept behind the **`FregeRepl`** and **`FregeInterpreter`** modules as can be seen from the type definitions in Listings 9 and 10. The **`Interpreter`** reads an **`InterpreterConfig`** and saves the computations in **`InterpreterState`** by combining the reader and state monad. The **`Repl`**, on the other hand, only saves the **`ReplEnv`** with the help of the state monad. However, the **`ReplEnv`** data type has both a config and state field, which become inputs for the **`Interpreter`**. As a result, the **`Repl`** can use the **`Interpreter`**, which I leverage in my `typeSignature` function shown in Listing 11.

```
type InterpreterState = StateT (MutableIO
↪  InterpreterClassLoader) StIO

newtype Interpreter result =
  Interpreter { un :: ReaderT InterpreterState
  ↪  InterpreterConfig result }
```

Listing 9: The **`Interpreter`** monad transformer type defined in the **`FregeInterpreter`** module.

```
data ReplEnv = ReplEnv
  { lastJavaGenSrc :: Maybe String
  , lastExternalScript :: Maybe String
  , opts :: ReplOpts
  , config :: InterpreterConfig
  , lineStart :: Int
  , state :: MutableIO InterpreterClassLoader
  }
data Repl a = Repl {un :: StateT ReplEnv IO a}
```

Listing 10: The `Repl` monad transformer type defined in the `FregeInterpreter` module.

```
typeSignature :: String -> ReplEnv -> IO (Maybe
↪  String, ReplEnv)
typeSignature fname env = Repl.run (evalType fname)
↪  env

evalType :: String -> Repl (Maybe String)
evalType expr = do
  env <- Repl.get
  res <- liftIO $ fst <$> Interpreter.run (typeof
  ↪  expr) env.config env.state
  case res of
    Left _     -> return Nothing
    Right typ -> return $ Just typ
```

Listing 11: The `typeSignature` function, which combines the two monad transformers `Repl` and `Interpreter` to return the type signature of an expression.

As a last step, I need to retrieve the result from the `typeSignature` function and return it in the LSP4J Java `hover` function to Visual Studio Code as introduced in Figure 2. Since the `typeSignature` function has type `IO`, I can only retrieve that in an impure way with the help of the `PreludeBase.TST.performUnsafe` function. As a result, all computations in Frege remain pure and become impure once they cross the Frege Java boundary.

VI. Discussion

Currently, the Frege language server provides two small features: It shows compiler warnings and errors when opening or saving a single `.fr` file (if there are any) and provides the type signature when hovering over the first word of a line. The server only supports the first word of a line because it is missing a representation of the source code in a data structure, which can be easily mapped to the abstract syntax tree of the Frege compiler. These two features make the Frege language server hardly usable because a typical language server not only supports hover and diagnostic features but many more such as autocompletion, refactoring, go to declaration and find usages. Furthermore, the hover request takes a long time until it returns the type signature.

However, the Frege language server provides a solid basis for future work. It uses the well-maintained LSP4J library to implement the current language server protocol `3.16.0` and possible future versions. It is well-tested, uses Github actions as a continuous delivery pipeline and directly integrates the Frege compiler through the `FregeRepl` and `FregeInterpreter` modules. This allows to develop the abovementioned additional language features to make the Frege language server not only usable in Visual Studio Code but for every language server protocol supporting editor.

VII. Conclusion

Developing a language server takes a lot of effort. A deep knowledge of the language's compiler or interpreter is required to build the needed language features. Hence, the language server protocol is a welcoming addition, which helps to make these features reusable across multiple editor clients.

In this experience report, I showed through multiple proof of concepts how a language server using the language server protocol can be developed iteratively. As a first step, I ported Microsoft's example language server to Java. Secondly, I built a Frege Gradle plugin, which compiles Frege files to Java allowing interoperability between Java and Frege. Lastly, I leveraged this interoperability to develop a diagnostic and hover feature powered by the Frege compiler, which is used in the Frege Visual Studio Code plugin.

References

[1] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the elipse ide?" *IEEE software*, vol. 23, no. 4, pp. 76–83, 2006.

[2] Microsoft, "Language server protocol," https://microsoft.github.io/language-server-protocol/, 2016, Accessed: 03-09-2021.

[3] T. Gagnaux, "Frege lsp server," https://github.com/tricktron/frege-lsp-server, 2021, Accessed: 03-09-2021.

[4] I. Wechsung, "The frege programming language (draft)," 2014.

[5] T. Gagnaux, "Frege vscode," https://github.com/tricktron/frege-vscode, 2021, Accessed: 03-09-2021.

[6] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language*. Addison Wesley Professional, 2005.

[7] StackOverflow, "2021 developer survey," https://insights.stackoverflow.com/survey/2021, 2021, Accessed: 28-08-2021.

[8] D. Geer, "Eclipse becomes the dominant java ide," *Computer*, vol. 38, no. 7, pp. 16–18, 2005.

[9] I. Wechsung, "fregide," https://github.com/Frege/eclipse-plugin, 2019, Accessed: 03-09-2021.

[10] P. Charles, R. M. Fuhrer, S. M. Sutton Jr, E. Duesterwald, and J. Vinju, "Accelerating the creation of customized, language-specific ides in eclipse," *ACM Sigplan Notices*, vol. 44, no. 10, pp. 191–206, 2009.

[11] Karnaukhov, Kirill and Surkov, Peter and Khudyakov, Jura, "Intellij idea plugin for frege language," https://github.com/IntelliJ-Frege/intellij-frege, 2021, Accessed: 03-09-2021.

[12] M. Madasamy, "Frege repl," https://github.com/Frege/frege-repl, 2019, Accessed: 03-09-2021.

[13] Microsoft, "Language server extension guide," https://code.visualstudio.com/api/language-extensions/language-server-extension-guide, 2021, Accessed: 03-09-2021.

[14] ——, "Vscode language server - node," https://github.com/Microsoft/vscode-languageserver-node, 2021, Accessed: 28-08-2021.

[15] T. E. Foundation, "Eclipse lsp4j," https://github.com/eclipse/lsp4j, 2021, Accessed: 03-09-2021.

[16] T. Gagnaux, "Lsp example with a java server and lsp4j," https://github.com/tricktron/vscode-extension-samples/tree/java-language-server-example/lsp-sample, 2021, Accessed: 28-08-2021.

[17] F. P. Miller, A. F. Vandome, and J. McBrewster, *Apache Maven*. Alpha Press, 2010.

[18] B. Muschko, *Gradle in action*. Simon and Schuster, 2014.

[19] Jetbrains, "The state of developer ecosystem 2021," https://www.jetbrains.com/lp/devecosystem-2021/java/, 2021, Accessed: 29-08-2021.

[20] Gradle, "The application plugin," https://docs.gradle.org/current/userguide/application_plugin.html#sec:application_tasks, 2021, Accessed: 28-08-2021.

[21] I. Wechsung, "Calling frege from java (from release 3.24 on)," https://github.com/Frege/frege/wiki/Calling-Frege-from-Java-(from-release-3.24-on)#thunks-and-boxes-fregerunthunk-fregerunbox, 2016, Accessed: 03-09-2021.

[22] Gradle, "The java plugin," https://docs.gradle.org/current/userguide/java_plugin.html#sec:java_tasks, 2021, Accessed: 28-08-2021.

[23] T. Gagnaux, "Frege gradle plugin," https://github.com/tricktron/frege-gradle-plugin, 2021, Accessed: 28-08-2021.

[24] M. P. Jones, “Functional programming with overloading and higher-order polymorphism,” in *International School on Advanced Functional Programming*. Springer, 1995, pp. 97–136.